Algorithmic Contributions to the Theory of Regular Chains

(Spine title: Algorithmic Contributions to the Theory of Regular Chains)

(Thesis format: Monograph)

by

Wei Pan

Graduate Program
in
Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada
January, 2011

THE UNIVERSITY OF WESTERN ONTARIO
THE SCHOOL OF GRADUATE AND POSTDOCTORAL STUDIES

CERTIFICATE OF EXAMINATION

Supervisor:	Examination committee:
Dr. Marc Moreno Maza	Dr. Mark Daley
	Dr. Jan Minac
	Dr. Jean-Louis Roch
	Dr. Roberto Solis-Oba

The thesis by

Wei Pan

entitled:

Algorithmic Contributions to the Theory of Regular Chains

is accepted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Date _____

_____
Chair of the Thesis Examination Board

Abstract

Regular chains, introduced about twenty years ago, have emerged as one of the major tools for solving polynomial systems symbolically. In this thesis, we focus on different algorithmic aspects of the study of regular chains, from theoretical questions to high-performance implementation issues.

The inclusion test for saturated ideals is a fundamental problem in the theory of regular chains. By introducing a notion of primitivity for regular chains, we show that a regular chain generates its saturated ideal if and only if it is primitive. As a consequence, we obtain new criteria for the inclusion test, which are effective in practice.

Computing regular greatest common divisors (GCDs) of two polynomials modulo a regular chain is one of the key routines in the various methods for solving polynomial systems by means of triangular decomposition. By revisiting the relations between polynomial subresultants and GCDs, we propose a novel bottom-up algorithm for this task, which improves the previous algorithm in a significant manner and creates opportunities for parallel execution.

In a third part, we present our efforts for accelerating the solving of bivariate polynomial systems the context of massively parallel architectures, such as graphics processor units (GPUs). Our building blocks like Fast Fourier transform (FFT) over finite fields and FFT-based subresultant chain constructions run faster by several orders of magnitude on GPUs than CPU counterparts.

Keywords. symbolic computation, solving polynomial system, regular chain, primitivity, inclusion test, regular GCD, subresultant, fast Fourier transform, GPU computing, CUDA.

Acknowledgments

It gives me great pleasure to thank all of those who made this thesis possible.

I am deeply grateful to my supervisor, Marc Moreno Maza for his constant support, guidance, ideas and encouragement during the past four years. Without his help, this thesis would not exist.

I want to thank my colleagues François Lemaire, Yuzhen Xie, Xin Li, Oleg Golubitsky, Changbo Chen, Sardar Anisul Haque, Liyun Li, Paul Vrbik, Rong Xiao for providing me help and for sharing their knowledge. Many thanks to all professors and students at ORCCA, where I spent this wonderful period of my life.

I want to thank our colleagues at Maplesoft, in particular Jürgen Gerhard and Clare So, for our cooperation on the `RegularChains` and `modpn` libraries and for their help during my internship at Maplesoft.

Many thanks to the members of my committee Prof. Mark Daley, Prof. Jan Minac, Prof. Jean-Louis Roch and Prof. Roberto Solis-Oba for their reading of this thesis and comments.

I am very grateful to my friends around the world for being my friends.

Special thanks to my entire family, my grandmother, grandmother-in-law, parents, parents-in-law, sisters, sisters-in-law, brothers-in-law, my wife, niece-in-law, my nephew, and nephews-in-law for their love.

To you, I dedicate this thesis, Yun.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Solving systems of polynomial equations is one of the most fundamental and most studied subjects in mathematical sciences. If the case of linear systems is theoretically well understood, the implementation of efficient linear solvers, in particular with very large and very sparse matrices, is still a popular research area.

The case of non-linear systems is, by essence, richer and more complex. Thus, research in this area covers theory, algorithms, implementation techniques and applications. Until the advent of computers, the theoretical part was obviously dominant. With the work of the algebraists of the early 20th century, Emmanuel Lasker, Emmy Nöther, David Hilbert, Bartel Leendert van der Waerden, Wolfgang Gröbner, the theoretical question of solving polynomial systems was, to some sense, solved by the notion of a primary decomposition of a polynomial ideal and that of the irreducible decomposition of an algebraic set. With the advent of computers, the quest for algorithms eligible to implementation began. In 1965, the PhD thesis of Bruno Buchberger (Student of W. Gröbner) brought the first such algorithm, based on the concept of a *Gröbner basis* together with a first computer implementation.

Before the work of Buchberger, several algorithms for solving certain types of algebraic or differential polynomial systems were proposed, but some details were often ignored since these algorithms could not be implemented or even performed manually (except on toy examples) due to their high complexity. Among them is the so-called *characteristic set method*, originally proposed for systems of ordinary differential equations, by Joseph Fels Ritt in the 1930's. The Chinese mathematician, Wen Tsún Wu, as he was employed in a semi-conductor plant during the cultural revolution, realized the power of computers. When he returned to the academia, he

bridged the gaps in the work of Ritt toward a practical algorithm and realized an implementation in the early 1980's.

Gröbner bases and characteristic sets are in fact one of the ways of representing the solutions of a polynomial system. Gröbner bases which, in some sense extend Gaussian elimination techniques to polynomial systems, have been widely studied and implemented since the PhD thesis of Buchberger. Today, some solvers based on Gröbner bases process large and difficult input systems, in particular when coefficients are computed modulo a prime number. Characteristic sets have required and still require more theoretical development. Indeed, they reveal more geometrical information from the input problem than Gröbner bases. One key step in this research effort on characteristic sets was the notion of a regular chain introduced independently in 1991 by Michael Kalkbrener (student of Bruno Buchberber) in his PhD thesis and, by Lu Yang and Jingzhong Zhang in [84].

These necessary theoretical advances explain why the realization of efficient solvers based on regular chains is still in its infancy. However, this is a very promising research direction for a variety of reasons. First, complexity results [76, 23] show that the space requirements for encoding the solutions of polynomial systems via regular chains can be regarded as nearly optimal, among all possible symbolic representations. Secondly, regular chains and their related techniques, by reducing multivariate arithmetic to univariate arithmetic, offer opportunities for making use of asymptotically fast algorithms (based on Fast Fourier Transform).

At the time of starting this thesis, several questions, combining theoretical, algorithmic and implementation aspects were hot topics in the theory of regular chains. As previous work in this area has illustrated, pursuing research simultaneously on these different aspects benefit to each of them.

1.2 Contributions of this thesis

In the multivariate polynomial ring $\mathbf{k}[\mathbf{x}] = \mathbf{k}[x_1, \ldots, x_n]$, over the field $\mathbf{k}$, we assume that the variables are ordered as $x_1 \prec \cdots \prec x_n$. Thus, any non-constant polynomial f can be viewed as a univariate polynomial in its largest variable, also called the main variable and denoted by $\mathsf{mvar}(f)$. The leading coefficient of f as a univariate polynomial in $\mathsf{mvar}(f)$ is called the initial of f, and denoted by $\mathsf{init}(f)$.

A *triangular set* T is defined as a set of non-constant multivariate polynomials in $\mathbf{k}[x_1, \ldots, x_n]$ such that polynomials in T have distinct main variables. The fundamental algebraic object associated to a triangular set is its *saturated ideal* $\mathsf{sat}(T)$

defined as

$$\mathsf{sat}(T) := \langle T \rangle : h^\infty = \{q \in \mathbf{k}[\mathbf{x}] \mid \exists e \geq 0 \text{ s.t. } h^e q \in \langle T \rangle\},$$

where h is the product of initials in T and $\langle T \rangle$ is the ideal generated by T in $\mathbf{k}[\mathbf{x}]$. Assuming T not empty, we write T as $T' \cup \{t\}$ where t is the polynomial in T with largest main variable. Then T is a *regular chain* if and only if T' is empty or, if T' is a regular chain and $\mathsf{init}(t)$ is not a zero-divisor modulo $\mathsf{sat}(T')$.

Primitive regular chains. One way of solving a polynomial system F is to decompose F into a finite set of regular chains $T_1, \ldots, T_e$ such that we have:

$$V(F) = V(\mathsf{sat}(T_1)) \cap \cdots \cap V(\mathsf{sat}(T_e)),$$

where $V(I)$ denotes the set of common solutions (over the algebraic closure of $\mathbf{k}$) of the polynomials in the ideal I. Due to the special shape of regular chains, many geometrical information is revealed by this decomposition, such as the possible emptiness of $V(F)$ or its dimension, etc.

Redundant (or superfluous) components are generated by all known methods decomposing the solutions of a polynomial system, whether these methods are purely symbolic or numeric. This is the case, in particular, for those methods based on regular chains. Once redundant components are removed, the size of the decomposition is often reduced in a significant manner; moreover a better insight on the geometry of those components can be obtained. For this reason, a fundamental problem in the theory of regular chains is that of the inclusion test for saturated ideals, that is, deciding whether $\mathsf{sat}(T) \subseteq \mathsf{sat}(U)$ holds for two regular chains T and U .[1]

Towards this problem, we have gained some results by studying the primitivity of polynomials over a commutative ring. In Section 2.4, we introduce the notion of *primitivity* for regular chains, which is a non-trivial generalization of the usual notion of primitivity for polynomials over unique factorization domains. Our main theorem in Section 2.4 can be stated as

Theorem 1. *Let $T \subset \mathbf{k}[x_1, \ldots, x_n]$ be a regular chain. Then T is primitive if and only if $\langle T \rangle = \mathsf{sat}(T)$ holds.*

[1] By means of Gröber basis computations, a set of generators of $\mathsf{sat}(T)$ can be computed, which in turn solves the inclusion test problem. However, the computations involved are extremely expensive. Our goal is to look for Gröbner-free algorithms for the inclusion test problem.

In Section 2.6, we present an efficient routine, Algorithm 1, to check whether a regular chain T is primitive. As an application of these theoretical results, a certain type of inclusion tests can be detected very efficiently as shown in Section 2.7.

This joint work with François Lemaire, Marc Moreno Maza and Yuzhen Xie is reported in [47] and the enhanced version is reported in [46].

A bottom-up regular GCD algorithm. The second objective of this thesis is to revisit key subroutines used in the Triade (TRIangular DEcomposition) algorithm of Marc Moreno Maza [65]. This algorithm is based on regular chains and is implemented in three computer algebra systems, including MAPLE, as part of the `RegularChains` library. At the core of the Triade algorithm, a generalized concept of polynomial GCDs extends the standard definition to the cases where coefficients are in a polynomial ring modulo a saturated ideal, as defined in Section 3.2.1.

Definition 1. *Let P, Q be polynomials in $\mathbf{k}[\mathbf{x}, y]$ and T be a regular chain in $\mathbf{k}[\mathbf{x}]$ such that $\mathsf{init}(P)$ and $\mathsf{init}(Q)$ are regular modulo $\mathsf{sat}(T)$. Then $G \in \mathbf{k}[\mathbf{x}, y]$ is a regular GCD of P, Q modulo $\mathsf{sat}(T)$, if the following conditions hold*

1. *the leading coefficient $\mathsf{lc}(G, y)$ of G in y is regular modulo $\mathsf{sat}(T)$,*
2. *there exist $u, v \in \mathbf{k}[\mathbf{x}, y]$ such that $G - uP - vQ \in \mathsf{sat}(T)$, and*
3. *if $\mathsf{deg}(G, y) > 0$ holds, then $P \in \mathsf{sat}(T \cup \{G\})$ and $Q \in \mathsf{sat}(T \cup \{G\})$ hold too.*

The calculation of these GCDs relies on the theory of subresultants which permit to work with polynomial coefficients. Indeed, the standard Euclidean algorithm is limited to univariate polynomials over a field. The original GCD procedure of the Triade algorithm has advantages like controlling the expression swell and splitting the computation for reducing algebraic complexity. However, it is a top-down "Euclidean-like" algorithm, each computation step relying on the results from the previous steps. This type of algorithms offer very limited opportunities for concurrent execution.

In Section 3.3 of Chapter 3, we closely examine the relations between the subresultant chain and the regular GCDs. The notion of a candidate regular GCD in Definition 5 is a very practical approximation of that of a regular GCD. According to Lemma 14, a candidate regular GCD is in fact a regular GCD, provided that $\mathsf{sat}(T)$ is a radical ideal.

Let $m = \min(\mathsf{deg}(P, y), \mathsf{deg}(Q, y))$. Then the subresultant chain of P, Q in y consists of m polynomials $S_0, \ldots, S_{m-1}$ in $\mathbf{k}[\mathbf{x}, y]$, defined formally in Section 3.2.3.

For each i, the degree of S_i in y is at most i, and hence the subresultant chain can be written as

$$\begin{aligned} S_{m-1} &= s_{m-1,m-1}y^{m-1} + s_{m-1,m-2}y^{m-2} + \cdots + s_{m-1,0} \\ S_{m-2} &= s_{m-2,m-2}y^{m-2} + \cdots + s_{m-2,0} \\ &\cdots \\ S_2 &= s_{22}y^2 + s_{21}y + s_{20} \\ S_1 &= s_{11}y + s_{10} \\ S_0 &= s_{00} \end{aligned}$$

where s_{ij} are polynomials in $\mathbf{k}[\mathbf{x}]$. Therefore, the subresultant chain can be viewed as a collection of $\frac{m(m+1)}{2}$ polynomials.

The distinction between the candidate regular GCD and the regular GCD is clarified by Lemma 13, based on which in Section 3.4 we present a novel algorithm RGSZR for computing regular GCDs. In this algorithm, the set of candidate regular GCDs is built by "regularizing" polynomials in order s_{00}, s_{11}, s_{10}, etc, from the bottom to the top of the subresultant chain. This step stops if coefficient s_{dd} is regular modulo $\mathsf{sat}(T)$, for some $0 \leq d < m$. The second step is to regularize coefficients s_{ii} with respect to $\mathsf{sat}(T)$ along the diagonal in the subresultant chain, for all $d < i < m$.

Comparing to the classical algorithms for computing regular GCDs, there are a number of advantages in the bottom-up algorithm RGSZR. First, the subresultant chain computation is separated from the computation of regular GCDs. In fact, The subresultant chain could be given by a black box, by which a subresultant S_i or a coefficient s_{ij} can be supplied whenever needed. This feature allows us to use modular methods for computing subresultant chains. We evaluate P and Q at sufficiently many points, and subresultants are only given by values. Whenever needed, S_i or s_{ij} are interpolated from their images. Secondly, we observe that, in practice, a regular GCD of P, Q has a lower degree than P and Q. Therefore only $O(m)$ coefficients s_{ij} are needed in this case.

In Chapter 4, we discuss the implementation techniques of computing subresultant chains over finite fields and analyze the complexity of the regular GCD algorithm under certain genericity assumptions. The subresultant chain of P and Q will be evaluated by means of (multi-dimensional) fast Fourier transforms (FFT). The univariate subresultant algorithm is conducted at each evaluation image, and the resulting data structure is called the subresultant cube (or SCube for short). This serves as the black box mentioned above: a subresultant S_i or a coefficient s_{ij} can be interpolated by inverse FFTs.

Assume that T is a zero-dimensional regular chain and that $\mathsf{sat}(T)$ is a radical ideal. Let $\{d_1, \ldots, d_n\}$ be the set of main degrees of polynomials in T, and let d_{n+1} be the minimum of $\mathsf{deg}(P, y)$ and $\mathsf{deg}(Q, y)$. The complexity result in Corollary 5 of Section 4.4 shows that the cost of computing regular GCDs of P, Q w.r.t T depends "quadratically" on the product of the degrees $d_1, \ldots, d_n, d_{n+1}$ and that the constant factor is approximately 2^n.

Under the same hypotheses, in [22], the authors achieve a running estimate which depends "linearly" (up to logarithmic factors) on the product of the degrees $d_1, \ldots, d_n, d_{n+1}$. However, their "exponential factor" is of the form c^n where $\mathsf{c} \geq 700$. Since practical values for $d_1, \ldots, d_n, d_{n+1}$ are often below the hundreds, in particular for d_{n+1} with n large, this suggests that the algorithms presented in Chapter 4 are more suitable for implementation than those of [22].

This joint work with Xin Li and Marc Moreno Maza is reported in [49]. Our improved regular GCD algorithm has been incorporated into the `RegularChain` library for arbitrary coefficient rings. Over finite fields, the C implementation in the library `modpn` demonstrates the high efficiency of our algorithm.

GPU acceleration. Solving polynomial systems involves intensive computations, as demonstrated in the resultant and GCD computations. The third objective of this thesis is to accelerate the symbolic computation algorithms by means of GPU (graphics processing unit) computing. In Appendix B, we briefly introduce the basics of GPU computing on Nvidia graphics cards, using CUDA (an acronym for Compute Unified Device Architecture).

Our first routine for GPU acceleration is fast Fourier transform (FFT) over finite fields, which is a basic routine for asymptotically fast algorithms for dense polynomial multiplications and divisions. It is also the starting point of the FFT based modular method for computing subresultants in Chapter 6. In Chapter 5 we report how to generate basic CUDA kernels for composing FFT formulas based on Kronecker product [73, 32]. In Section 5.6, we compare our implementation and optimization issues of Cooley-Tukey FFT and Stockham FFT. Comparing to the optimized serial C code in the library `modpn`, our CUDA Stockham FFT achieves a speedup factor of 37 for large input sizes. This joint work with Marc Moreno Maza is reported in [57].

In Chapter 6, we describe how to accelerate the FFT based subresultant chain computation for multivariate polynomials over finite fields. The multivariate problem is turned into that of bivariate or trivariate ones, by means of Kronecker's substitutions. Due to the special requirements of the FFT based modular method, the leading

coefficients of the input polynomials cannot vanish at any evaluation point. In Section 6.4, we present and analyze the method of using linear translations to enlarge the applicable range of FFT based methods. In Section 6.5 and Section 6.6, we revisit Brown's subresultant algorithm, estimate the size and cost of computing the subresultant evaluation cube. Experimental results reported in Section 6.7, show a significant speedup factor between our fine-grained implementation and the `modpn` counterpart. Our CUDA routines are approximately 25 times faster in the bivariate case and approximately 47 times faster in the trivariate case.

Chapter 2

Primitive Regular Chain

2.1 Introduction

Triangular decompositions are one of the most studied techniques for solving polynomial systems symbolically. Invented by J.F. Ritt in the early 30's for systems of differential polynomials, their stride started in the late 80's with the method of [83] dedicated to algebraic systems. Different concepts and algorithms extended the work of Wu. In the early 90's, the notion of a *regular chain*, introduced independently by [41] and by [84], led to important algorithmic discoveries.

In Kalkbrener's vision, regular chains are used to represent the generic zeros of the irreducible components of an algebraic variety. In the original work of Yang and Zhang, they are used to decide whether a hypersurface intersects a quasi-variety (given by a regular chain). Regular chains have, in fact, several interesting properties and are the key notion in many algorithms for decomposing systems of algebraic or differential equations.

Regular chains have been investigated in many papers, among them are those of [6], [16] and [42]. Several surveys [8, 39] are also available on this topic. The abundant literature on the subject can be explained by the many equivalent definitions of a regular chain. Actually, the original formulation of Kalkbrener is quite different from that of Yang and Zhang. In the papers by [14] and [82], the authors provide bridges between the point of view of Kalkbrener and that of Yang and Zhang.

The key algebraic object associated with a regular chain is its *saturated ideal*. Let us review its definition. Let $\mathbf{k}$ be a field and $x_1 \prec \cdots \prec x_n$ be ordered variables. For a regular chain $T \subset \mathbf{k}[x_1, \ldots, x_n]$, the saturated ideal of T, denoted by $\mathsf{sat}(T)$ is defined by $\mathsf{sat}(T) := \langle T \rangle : h^\infty$, where h is the product of the initial polynomials of T. (The next section contains a detailed review of these notions.) Given a polynomial

$p \in \mathbf{k}[x_1, \ldots, x_n]$, the memberships $p \in \mathsf{sat}(T)$ and $p \in \sqrt{\mathsf{sat}(T)}$ can be decided by means of pseudo-divisions and GCD computations, respectively. One should observe that these computations can be achieved without computing a system of generators of $\mathsf{sat}(T)$. In some sense, the regular chain T is a "black box representation" of $\mathsf{sat}(T)$ since the assertions $p \in \mathsf{sat}(T)$ and $p \in \sqrt{\mathsf{sat}(T)}$ can be evaluated without using an explicit representation of $\mathsf{sat}(T)$.

Being able to compute a system of generators of $\mathsf{sat}(T)$ remains, however, a fundamental question. For instance, given a second regular chain $U \subset \mathbf{k}[x_1, \ldots, x_n]$, the only general method to decide the inclusion $\mathsf{sat}(T) \subseteq \mathsf{sat}(U)$ goes through the computation of a system of generators of $\mathsf{sat}(T)$ by means of Gröbner bases. Unfortunately, such computations can be expensive [see 7] whereas one would like to obtain an inclusion test which could be used intensively in order to remove redundant components when computing the triangular decompositions of Kalkbrener's algorithm or those arising in differential algebra. Note that for other kinds of triangular decompositions, such as those of [65] and [82], this question has been solved in [15].

Therefore, testing the inclusion $\mathsf{sat}(T) \subseteq \mathsf{sat}(U)$ without Gröbner basis computation is a very important question in practice. Moreover, this can be regarded as an *algebraic version* of the Ritt problem in differential algebra. One case presents no difficulties: if $\mathsf{sat}(T)$ is a zero-dimensional ideal, the product of the initial polynomials of T is invertible modulo $\langle T \rangle$ [see 67, Proposition 5] and thus T generates $\mathsf{sat}(T)$. In this case the inclusion test for saturated ideals reduces to the membership problem mentioned above.

In positive dimension, however, the ideal $\mathsf{sat}(T)$ could be strictly larger than that generated by T. Consider for instance $n = 4$ and $T = \{x_1x_3 + x_2, x_2x_4 + x_1\}$, we have

$$\langle T \rangle \;=\; \langle x_1, x_2 \rangle \cap \langle x_1x_3 + x_2, -x_3x_4 + 1 \rangle.$$

Thus, we have

$$\mathsf{sat}(T) \;=\; \langle T \rangle : (x_1x_2)^\infty \;=\; \langle x_1x_3 + x_2, -x_3x_4 + 1 \rangle.$$

In this article, we give a necessary and sufficient condition for the equality $\langle T \rangle = \mathsf{sat}(T)$ to hold. Looking at the above example, one can feel that the ideal $\langle x_1, x_2 \rangle$ can be regarded as a "sort of content" of the ideal $\langle T \rangle$, which is discarded when computing $\mathsf{sat}(T)$. We observe also that the polynomials $x_1x_3 + x_2$ and $x_2x_4 + x_1$ are primitive in $(\mathbf{k}[x_1, x_2])[x_3]$ and $(\mathbf{k}[x_1, x_2])[x_4]$ respectively. Thus, the "usual notion"

of primitivity (for a univariate polynomial over a UFD) is not sufficient to guarantee the equality $\langle T \rangle = \mathsf{sat}(T)$. This leads us to the following two definitions.

Let R be a commutative ring with unity. We say that a non-constant polynomial $p = a_e x^e + \cdots + a_0 \in R[x]$ is *weakly primitive* if for any $\beta \in R$ such that a_e divides $\beta a_{e-1}, \ldots, \beta a_0$ then a_e divides β as well. This notion and its relations with similar concepts are discussed in Sections 2.3, 2.4, and 2.5.

We say that the regular chain $T = \{p_1, \ldots, p_m\}$ is *primitive* if for all $1 \leq k \leq m$, the polynomial p_k is weakly primitive in $R[x_j]$, where x_j is the main variable of p_k and R is the residue class ring $\mathbf{k}[x_1, \ldots, x_{j-1}]/\langle p_1, \ldots, p_{k-1} \rangle$.

The first main result of this chapter is the following: *the regular chain T generates its saturated ideal if and only if T is primitive.* This result, generalizing the concept of primitivity from univariate polynomials to regular chains, is established in Section 2.4.

Looking at regular chains from the point of view of regular sequences, we obtain our second main result: an algorithm to decide whether a regular chain generates its saturated ideal or not. The pseudo-code and its proof are presented in Section 2.6. This algorithm relies on a procedure for computing triangular decompositions. However, being applied to input systems which are regular sequences and "almost regular chains", this procedure reduces simply to an iterated resultant computation. As a result, the proposed algorithm performs very well in practice and is Gröbner basis free. In Section 2.8 we report on experimentation, where we confirm the efficiency of this algorithm. Meanwhile, we observe that primitive regular chains are often present in the output of triangular decompositions.

Section 2.7, which is a new development w.r.t. our ISSAC paper [47], proposes several criteria for testing the inclusion of saturated ideals. We point out that the notion of primitivity of regular chains provides a helpful tool for dealing with this question in practice. Section 2.9, which is also enhanced w.r.t. [47], offers concluding remarks and open problems.

This joint work with François Lemaire, Marc Moreno Maza and Yuzhen Xie is reported is reported in [46].

2.2 Preliminaries

2.2.1 Triangular set and regular chain

We denote by $\mathbf{k}[\boldsymbol{x}]$ the ring of multivariate polynomials with coefficients in a field $\mathbf{k}$ and with ordered variables $\boldsymbol{x} = x_1 \prec \cdots \prec x_n$. For a non-constant polynomial $p \in \mathbf{k}[\boldsymbol{x}]$,

the greatest variable appearing in p is called *main variable*, denoted by $\mathsf{mvar}(p)$. We regard p as a univariate polynomial in its main variable. The degree, the leading coefficient, the leading monomial and the reductum of p as a univariate polynomial in $\mathsf{mvar}(p)$ are called *main degree*, *initial*, *rank* and *tail* of p; they are denoted by $\mathsf{mdeg}(p)$, $\mathsf{init}(p)$, $\mathsf{rank}(p)$ and $\mathsf{tail}(p)$ respectively. Thus we have $p = \mathsf{init}(p)\mathsf{rank}(p) + \mathsf{tail}(p)$.

Let R be a commutative ring with unity and F be a subset of R. Denote by $\langle F\rangle$ the *ideal* it generates, by $\sqrt{\langle F\rangle}$ the radical of $\langle F\rangle$, and by $R/\langle F\rangle$ the *residue class ring* of R with respect to $\langle F\rangle$. For an element p in R, we say that p is zero modulo $\langle F\rangle$ if p belongs to $\langle F\rangle$, that is, p is zero as an element in $R/\langle F\rangle$. An element $p \in R$ is a *zerodivisor* modulo $\langle F\rangle$, if there exists $q \in R$ such that $p \notin \langle F\rangle$ and $q \notin \langle F\rangle$ but $pq \in \langle F\rangle$. We say that p is *regular* modulo $\langle F\rangle$ if it is neither zero, nor a zerodivisor modulo $\langle F\rangle$. Furthermore, p is *invertible* in R if there exists a $q \in R$ such that $p\,q = 1$.

Example 1. *Consider the polynomials in* $\mathbf{k}[x_1, x_2, x_3]$

$$p_1 = x_2^2 - x_1^2, p_2 = (x_2 - x_1)x_3 \quad \text{and} \quad p_3 = x_2x_3^3 - x_1.$$

The above notions are illustrated in the following table.

	mvar	init	mdeg	rank	tail
p_1	x_2	1	2	x_2^2	$-x_1^2$
p_2	x_3	$x_2 - x_1$	1	x_3	0
p_3	x_3	x_2	3	x_3^3	$-x_1$

The initial $x_2 - x_1$ of p_2 is a zerodivisor modulo $\langle p_1\rangle$, since $(x_2 + x_1)(x_2 - x_1)$ is in $\langle p_1\rangle$, while neither $x_2 + x_1$ nor $x_2 - x_1$ belongs to $\langle p_1\rangle$. However, the initial x_2 of p_3 is regular modulo $\langle p_1\rangle$.

In what follows, we recall the notions of regular chain and saturated ideal, which are the main objects in our study.

A set T of non-constant polynomials in $\mathbf{k}[\boldsymbol{x}]$ is called a *triangular set*, if for all $p, q \in T$ with $p \neq q$ we have $\mathsf{mvar}(p) \neq \mathsf{mvar}(q)$. For a nonempty triangular set T, we define the *saturated ideal* $\mathsf{sat}(T)$ of T to be the ideal $\langle T\rangle : h^\infty$, that is,

$$\mathsf{sat}(T) := \langle T\rangle : h^\infty = \{q \in \mathbf{k}[\boldsymbol{x}] \mid \exists e \in \mathbb{Z}_{\geq 0} \text{ s.t. } h^e q \in \langle T\rangle\},$$

where h is the product of the initials of the polynomials in T. The empty set is also regarded as a triangular set, whose saturated ideal is the trivial ideal $\langle 0\rangle$.

One way of solving (or decomposing) a polynomial set $F \subseteq \mathbf{k}[\boldsymbol{x}]$ is to compute triangular sets $T_1, \ldots, T_e \subseteq \mathbf{k}[\boldsymbol{x}]$ such that

$$\sqrt{\langle F \rangle} = \sqrt{\mathsf{sat}(T_1)} \cap \cdots \cap \sqrt{\mathsf{sat}(T_e)}.$$

It is thus desirable to require $\mathsf{sat}(T_1), \ldots, \mathsf{sat}(T_e)$ to be proper ideals. This observation has led to the notion of a regular chain which was introduced independently in [41, 84].

Definition 2 (Regular chain)**.** *Let T be a triangular set in $\mathbf{k}[\boldsymbol{x}]$. If T is empty, then it is a regular chain. Otherwise, let p be the polynomial of T with the greatest main variable and let C be the set of other polynomials in T. We say that T is a* regular chain*, if C is a regular chain and $\mathsf{init}(p)$ is regular modulo $\mathsf{sat}(C)$.*

In commutative algebra [see 29] there is a closely related concept called *regular sequence* which is a sequence $r_1, \ldots, r_s$ of nonzero elements in the ring $\mathbf{k}[\boldsymbol{x}]$ satisfying

1. $\langle r_1, \ldots, r_s \rangle$ is a proper ideal of $\mathbf{k}[\boldsymbol{x}]$;
2. r_i is regular modulo $\langle r_1, \ldots, r_{i-1} \rangle$, for each $2 \leq i \leq s$.

When we sort polynomials in a regular chain by increasing main variable, the following example says that the resulting sequence may not be a regular sequence of $\mathbf{k}[\boldsymbol{x}]$.

Example 2. *Let $T = \{t_1, t_2\}$ be a triangular set in $\mathbf{k}[x_1, x_2, x_3]$ with $t_1 = x_1x_2$ and $t_2 = x_1x_3$. Then $\{t_1\}$ is a regular chain with $\mathsf{sat}(\{t_1\}) = \langle x_1x_2 \rangle : x_1^\infty = \langle x_2 \rangle$. Since $\mathsf{init}(t_2) = x_1$ is regular modulo $\mathsf{sat}(\{t_1\})$, the triangular set T is a regular chain with*

$$\mathsf{sat}(T) = \langle x_1x_2, x_1x_3 \rangle : x_1^\infty = \langle x_2, x_3 \rangle.$$

However, t_1, t_2 is not a regular sequence since $t_2 = x_1x_3$ is not regular modulo $\langle x_1x_2 \rangle$. Here, the saturation operation discards *the content introduced by the initials.*

2.2.2 Properties of regular chains

We recall several important results on regular chains and saturated ideals, which will be used throughout this chapter. Pseudo-division and iterated resultant are fundamental tools in this context.

Let p and q be polynomials of $\mathbf{k}[\boldsymbol{x}]$, with $q \notin \mathbf{k}$. Denote by $\mathsf{prem}(p, q)$ and $\mathsf{pquo}(p, q)$ the *pseudo-remainder* and the *pseudo-quotient* of p by q, regarding p and q as univariate polynomials in $x = \mathsf{mvar}(q)$. Using these notations, we have

$$\mathsf{init}(q)^e p = \mathsf{pquo}(p, q)q + \mathsf{prem}(p, q), \tag{2.1}$$

where $e = \max\{\mathsf{deg}(p, x) - \mathsf{deg}(q, x) + 1, 0\}$; moreover either $r := \mathsf{prem}(p, q)$ is null or $\mathsf{deg}(r, x) < \mathsf{deg}(q, x)$. Pseudo-division generalizes as follows given a polynomial p and a regular chain T :

$$\mathsf{prem}(p, T) = \begin{cases} p & \text{if } T = \emptyset, \\ \mathsf{prem}(\mathsf{prem}(p, t), T') & \text{if } T = T' \cup \{t\}, \end{cases}$$

where t is the polynomial in T with greatest main variable. We have the *pseudo-division formula* [83]: there exist non-negative integers $e_1, \ldots, e_s$ and polynomials $q_1, \ldots, q_s$ in $\mathbf{k}[\boldsymbol{x}]$ such that

$$h_1^{e_1} \cdots h_s^{e_s} p = \sum_{i=1}^{s} q_i t_i + \mathsf{prem}(p, T), \tag{2.2}$$

where $T = \{t_1, \ldots, t_s\}$ and $h_i = \mathsf{init}(t_i)$, for $1 \leq i \leq s$.

We denote by $\mathsf{res}(p, q)$ the *resultant* of p and q regarding them as univariate polynomials in $\mathsf{mvar}(q)$. Note that $\mathsf{res}(p, q)$ may be different from $\mathsf{res}(q, p)$, if they have different main variables. For a polynomial p and a regular chain T, we define the *iterated resultant* of p w.r.t. T, denoted by $\mathsf{iterRes}(p, T)$, as follows:

$$\mathsf{iterRes}(p, T) = \begin{cases} p & \text{if } T = \emptyset, \\ \mathsf{iterRes}(\mathsf{res}(p, t), T') & \text{if } T = T' \cup \{t\}, \end{cases}$$

where t is the polynomial in T with greatest main variable.

Theorem 2. *For a regular chain T and a polynomial p we have:*

1. *p belongs to $\mathsf{sat}(T)$ if and only if $\mathsf{prem}(p, T) = 0$,*
2. *p is regular modulo $\mathsf{sat}(T)$ if and only if $\mathsf{iterRes}(p, T) \neq 0$,*
3. *p is a zerodivisor modulo $\mathsf{sat}(T)$ if and only if $\mathsf{iterRes}(p, T) = 0$ and $\mathsf{prem}(p, T) \neq 0$.*

For the proofs, we refer to [6] for item (1), and to [82, 14] for item (2). Item (3) is a direct consequence of (1) and (2).

Remark 1. *Theorems 2 and 3 highlight the structure of the associated primes of $\mathsf{sat}(T)$ which makes regularity test easier than with an arbitrary polynomial ideal. In general, deciding if a polynomial p is regular modulo an ideal I is equivalent to checking if p does not belong to any associated primes of I.*

An ideal in $\mathbf{k}[\boldsymbol{x}]$ is *unmixed*, if all its associated primes have the same dimension. In particular, an unmixed ideal has no embedded associated primes.

Theorem 3. *Let $T = C \cup \{t\}$ be a regular chain in $\mathbf{k}[\boldsymbol{x}]$ with t having greatest main variable in T. The following properties hold:*

1. $\mathsf{sat}(T)$ *is an unmixed ideal with dimension* $n - |T|$,
2. $\mathsf{sat}(T \cap \mathbf{k}[x_1, \ldots, x_i]) = \mathsf{sat}(T) \cap \mathbf{k}[x_1, \ldots, x_i]$,
3. $\mathsf{sat}(T) = \langle \mathsf{sat}(C) \cup \{t\} \rangle : \mathsf{init}(t)^\infty$.

For the proofs, we refer to [8, 16] for item (1), to [6] for item (2), and to [43] for item (3). From (1), we deduce that the saturated ideal of a regular chain T consisting of n polynomials has dimension 0.

2.3 Primitivity of polynomials

In this section, we introduce the notion of weak primitivity of a polynomial in a general univariate polynomial ring, and then present several of its properties.

The following Lemma 1 may be seen as a generalization of Gauss's lemma over an arbitrary commutative ring with unity. It will be used in the proof of our main theorem. We found that this lemma is not new and can be deduced from the Dedekind-Mertens Lemma (See [4, 20, 80] and the references therein). For the sake of reference, we include our direct proof here. In the sequel, the ring R is a commutative Noetherian ring with unity. We say that p divides q, denoted by $p \mid q$, if there exists r such that $q = p\,r$ holds.

Lemma 1. *Let $p = \sum_{i=0}^{m} a_i y^i$ and $q = \sum_{i=0}^{n} b_i y^i$ be polynomials in $R[y]$ with $\mathsf{deg}(p) = m \geq 0$ and $\mathsf{deg}(q) = n \geq 0$. Then for each $h \in R$,*

(*i*) *$h \mid pq$ implies $h \mid b_0 a_i^{n+1}$ for $0 \leq i \leq m$,*

(*ii*) *$h \mid pq$ implies $h \mid b_n a_i^{n+1}$ for $0 \leq i \leq m$.*

Proof. First, we prove (i). Considering first the special case $m = 0$, we observe that $h \mid pq$ implies $h \mid a_0 b_0$ and the conclusion follows. Now we assume that $m > 0$ holds.

For $i = 0$, the claim is also clear, for the same reason as the case $m = 0$. For $1 \leq i \leq m$, we introduce the polynomials A_i and B_i below in order to simplify our

expressions:

$$A_i = \sum_{j=0}^{i-1} a_j y^j, \text{ and } B_i = -\sum_{j=i}^{m} a_j y^j. \tag{2.3}$$

Clearly, we have $p = A_i - B_i$. The key observation is to consider the polynomial $\tilde{p} = A_i^{n+1} - B_i^{n+1}$, as suggested by the forms of our claims. To avoid talking about the degree of a zero polynomial, we assume that both A_i^{n+1} and B_i^{n+1} are nonzero polynomials.

According to the construction of A_i and B_i in (2.3), we have the following degree estimates:

$$\mathsf{deg}(A_i^{n+1}) \leq \mathsf{deg}(A_i)(n+1) \leq (i-1)(n+1), \tag{2.4}$$

$$\text{trdeg}(B_i^{n+1}) \geq \text{trdeg}(B_i)(n+1) \geq i(n+1), \tag{2.5}$$

where trdeg$(\cdot)$ denotes the trailing degree, that is, the degree of the term with lowest degree in a polynomial. Therefore there is no term cancellation between A_i^{n+1} and B_i^{n+1}. With the assumption that A_i and B_i nonzero, the polynomial $\tilde{p}$ is nonzero too. Now we write $\tilde{p}$ in the form

$$\tilde{p} = (A_i - B_i)(A_i^n + \cdots + B_i^n) = p(A_i^n + \cdots + B_i^n).$$

It follows that $p \mid \tilde{p}$ holds. Therefore $h \mid \tilde{p}q$ holds since we have $h \mid pq$. Observe now that if qA_i^{n+1} is nonzero, then

$$\mathsf{deg}(qA_i^{n+1}) \leq (i-1)(n+1) + n < i(n+1). \tag{2.6}$$

Similarly, if qB_i^{n+1} is nonzero, then its trailing degree is bounded

$$\text{trdeg}(qB_i^{n+1}) \geq i(n+1). \tag{2.7}$$

Combining (2.6) with (2.7), we know that in $q\tilde{p} = qA_i^{n+1} - qB_i^{n+1}$, the polynomial qA^{n+1} only contributes to terms with degree smaller than $i(n+1)$. Thus we have

$$\mathsf{coeff}(q\tilde{p}, y^{i(n+1)}) = \mathsf{coeff}(-qB_i^{n+1}, y^{i(n+1)}) = b_0 a_i^{n+1} \tag{2.8}$$

which implies $h \mid b_0 a_i^{n+1}$, as desired.

Now we handle the special cases where $A_i^{n+1} = 0$ and $B_i^{n+1} = 0$. It is easy to

see that $A_i^{n+1} = 0$ does not affect the proof above. When $B_i^{n+1} = 0$, simply we have $a_i^{n+1} = 0$, and then the claim is also clear.

Finally, we prove (ii). Let $P = y^m p(1/y)$ and $Q = y^n q(1/y)$. Since $h \mid pq$, h will also divide $PQ = y^{m+n}(pq)(1/y)$. Assume that

$$a_0 = \cdots = a_{r-1} = 0, a_r \neq 0,$$
$$b_0 = \cdots = b_{s-1} = 0, b_s \neq 0.$$

Then $r \leq m$ and $s \leq n$ hold. According to (i), for any $r \leq i \leq m$, $h \mid b_n a_i^{s+1}$. It follows that $h \mid b_n a_i^{n+1}$ for any $0 \leq i \leq m$. □

Definition 3. *Let $p = a_0 + \cdots + a_e x^e \in R[x]$ with $e \geq 1$. The polynomial p is* strongly primitive *if the ideal generated by $\{a_0, \ldots, a_e\}$ is the whole ring R. The polynomial p is* weakly primitive *if for any $\beta \in R$ such that $a_e \mid \beta a_i$ holds for all $0 \leq i \leq e-1$, we have $a_e \mid \beta$ as well.*

Proposition 1. *Strong primitivity implies weak primitivity.*

Proof. We use the same notation as in Definition 3. Let p be strongly primitive. Then there exist $c_e, \ldots, c_0 \in R$ such that $c_e a_e + \cdots + c_0 a_0 = 1$. Let $\beta \in R$ such that for $0 \leq j \leq e-1$, we have $a_e \mid \beta a_j$. Then there exist $d_0, \ldots, d_{e-1} \in R$ such that $a_e d_j = \beta a_j$. Since $\beta c_e a_e + \cdots + \beta c_0 a_0 = \beta$, we have $a_e(\beta c_e + d_{e-1} c_{e-1} \cdots + d_0 c_0) = \beta$. Thus, we have $a_e \mid \beta$, and therefore p is weakly primitive. □

Remark 2.

(1) *If any a_i is invertible, then p is strongly primitive and so it is weakly primitive. As a particular case, p is weakly primitive if one of its coefficients is a nonzero constant of a field.*

(2) *Weak primitivity does not imply strong primitivity. For example, let $R = \mathbb{Z}[t]$ and $p = tx + 2 \in \mathbb{Z}[t][x]$. Then p is not strongly primitive, since $\langle t, 2\rangle \neq \langle 1\rangle_R$. In $R[x]$, the polynomial p is weakly primitive. If $t \mid 2\beta$, then $t \mid \beta$ must hold.*

(3) *The definition of strongly primitive does not depend on the* order *of the coefficients in p. However, the definition of weakly primitive relies on it. Indeed, let $R = \mathbb{Z}_4[t]$, $p = \bar{2}x + t$ and $q = tx + \bar{2}$. Then we have*

   (a) *p is weakly primitive in $R[x]$. For any $\beta \in R[x]$, if $\bar{2} \mid t\beta$ then $\bar{2} \mid \beta$.*

   (b) *q is not weakly primitive in $R[x]$. Let $\beta = t + \bar{2} \in R[x]$. Then we have $t \mid \bar{2}(t + \bar{2}) = \bar{2}t$, and $t \nmid (t + 2)$.*

(4) *Weak primitivity may not be extended. That is to say, if p is weakly primitive, assuming that $\mathsf{deg}(p) = e > 0$, then $\bar{p} = p + qx^{e+1}$ may not be weakly primitive. For example, let $R = \mathbb{Z}_4[t]$, $p = \bar{2}x + t$ and $\bar{p} = p + tx^2 = tx^2 + \bar{2}x + t$. Then p is weakly primitive, and $\bar{p}$ is not weakly primitive. Indeed taking $\beta = t + \bar{2}$, we have $t \mid t\beta$ and $t \mid \bar{2}\beta$, but $t \nmid \beta$.*

According to Proposition 2 the notion of weak primitivity turns out to be a generalization of the ordinary notion of primitivity (the gcd of the coefficients of a univariate polynomial is 1).

Proposition 2. *Let R be a UFD and $p = \sum_{i=0}^{e} a_i x^i \in R[x]$ with $a_e \neq 0$ and $e \geq 1$. Then, the following statements are equivalent*

(*i*) *p is weakly primitive in $R[x]$.*

(*ii*) $\mathsf{content}(p) := \mathsf{gcd}(a_0, \ldots, a_e) = 1$.

Proof. We prove $(i) \Rightarrow (ii)$. Assume that $\mathsf{gcd}(a_0, \ldots, a_e) \neq 1$. Then there is a prime factor f of $\mathsf{gcd}(a_0, \ldots, a_e)$. Let $\beta = a_e/f$. Then $a_e \mid \beta a_i$, for $0 \leq i \leq e-1$. Since $a_e \nmid \beta$, p is not weakly primitive, a contradiction.

We prove $(ii) \Rightarrow (i)$. Assume that there exists $\beta \in R$ such that

$$\forall\,(\ 0 \leq j \leq e-1) \quad a_e \mid \beta a_j \quad \text{ and } \quad a_e \nmid \beta.$$

Then $a_e \mid \mathsf{content}(\beta p) = \beta \mathsf{content}(p)$. Since $a_e \nmid \beta$, some prime factor f of a_e divides $\mathsf{content}(p)$, a contradiction. □

The following property on weak primitivity will be used in the next section. It states the following fact: if one raises each coefficient of a weakly primitive polynomial p to some power, then the resulting polynomial is still weakly primitive. To avoid the cancellation of the leading coefficient of p, we assume that this coefficient is a regular element of the ground ring.

Proposition 3. *Let $p = \sum_{i=0}^{e} a_i x^i \in R[x]$ with a_e being regular in R, and $\{n_i \mid 0 \leq i \leq e\}$ be a set of non-negative integers. Define $q = \sum_{i=0}^{e} a_i^{n_i} x^i$. Then if p is weakly primitive, q is also weakly primitive.*

The proof directly follows from the following two lemmas.

Lemma 2. *Let $p = a_0 + \cdots + a_e x^e \in R[x]$ with a_e being regular in R and n be a non-negative integer. If p is weakly primitive, then $p_n = a_0 + \cdots + a_{e-1}x^{e-1} + a_e^n x^e$ is also weakly primitive.*

Proof. By induction on $n \geq 0$. The case $n = 0$ follows from Remark 2. So we assume that the claim is true for $n-1$, that is, p_{n-1} is weakly primitive, with $n \geq 1$. Let $\beta \in R$ such that $a_e^n \mid a_i\beta$, for $0 \leq i \leq e-1$. There exist $h_0, \ldots, h_{e-1} \in R$ such that we have

$$a_e^n h_i = a_i\beta, \quad 0 \leq i \leq e-1. \tag{2.9}$$

Since p_{n-1} is weakly primitive and since we have $a_e^{n-1} \mid a_i\beta$, we deduce $a_e^{n-1} \mid \beta$, that is, there exists $h' \in R$ such that

$$a_e^{n-1} h' = \beta. \tag{2.10}$$

With (2.9) and (2.10) we have $a_e^n h_i = a_i a_e^{n-1} h'$, and then $a_e h_i = a_i h'$, since a_e is regular. Hence $a_e \mid a_i h'$. By the weak primitivity of p, $a_e \mid h'$ holds, that is, there exists $h'' \in R$ such that

$$a_e h'' = h'. \tag{2.11}$$

By (2.10) and (2.11) we have $a_e^n h'' = \beta$. So $a_e^n \mid \beta$ and p_n is weakly primitive. □

Lemma 3. *Let $p = a_0 + \cdots + a_e x^e \in R[x]$ with $a_e \neq 0$ and n be a non-negative integer. Let j be an index such that $0 \leq j \leq e-1$. Define $q = a_0 + \cdots + a_j^n x^j + \cdots + a_e x^e = p + (a_j^n - a_j)x^j$. If p is weakly primitive, then q is also weakly primitive.*

Proof. The claim is clear if $n = 0$, so we assume $n \geq 1$. Let $\beta \in R$ such that, for $0 \leq i \leq e-1$ and $i \neq j$

$$a_e \mid a_i\beta, \text{ and } a_e \mid a_j^n\beta. \tag{2.12}$$

We prove that $a_e \mid \beta$ holds. We have, for $0 \leq i \leq e-1$ and $i \neq j$

$$a_e \mid a_i(a_j^{n-1}\beta), \text{ and } a_e \mid a_j(a_j^{n-1}\beta).$$

Define $\beta' = a_j^{n-1}\beta$. Hence $a_e \mid \beta'$ holds, since p is weakly primitive. With (2.12), for $0 \leq i \leq e-1$ and $i \neq j$ we have

$$a_e \mid a_i\beta, \text{ and } a_e \mid a_j^{n-1}\beta. \tag{2.13}$$

We deduce that $a_e \mid a_j^{n-2}\beta$ holds. Continuing in this manner, we reach $a_e \mid \beta$. Thus q is also weakly primitive. □

2.4 Primitive regular chain

In this section, we generalize the notion of primitivity to any regular chain T. Then we prove that $\mathsf{sat}(T) = \langle T \rangle$ holds if and only if T is primitive.

Definition 4. *Let $T = \{p_1, \ldots, p_m\} \subset \mathbf{k}[\boldsymbol{x}] = \mathbf{k}[x_1, \ldots, x_n]$ be a regular chain with $\mathsf{mvar}(p_1) \prec \cdots \prec \mathsf{mvar}(p_m)$. We say that T is* primitive *if for all $1 \leq k \leq m$, p_k is weakly primitive in $R[x_j]$ where $x_j = \mathsf{mvar}(p_k)$ and*

$$R = \mathbf{k}[x_1, \ldots, x_{j-1}]/\langle p_1, \ldots, p_{k-1} \rangle.$$

Proposition 4 (Base case of Theorem 4)**.** *Let $p = a_e x^e + \cdots + a_0 \in \mathbf{k}[\boldsymbol{y}][x]$ and $c = \mathsf{gcd}_{\mathbf{k}[\boldsymbol{y}]}(a_0, \ldots, a_e)$, where $e \geq 1$ and $\boldsymbol{y}$ is a finite set of variables. Then we have $\langle p \rangle = \langle p \rangle : a_e^\infty \iff c = 1$.*

Proof. First we prove that $\langle p \rangle \subsetneq \mathsf{sat}(p) := \langle p \rangle : a_e^\infty$ if $c \neq 1$. Denote $\bar{p} = p/c$. Then $a_e \bar{p} = a_e p/c \in \langle p \rangle$, hence $\bar{p} \in \mathsf{sat}(p)$. Assume that $\bar{p}$ is in $\langle p \rangle$. Then there exists $q \in \mathbf{k}[\boldsymbol{y}][x]$ such that $p/c = \bar{p} = pq$. It follows that $qc = 1$ which is a contradiction since $c \notin \mathbf{k}$. Therefore $\bar{p}$ is in $\mathsf{sat}(p)$ but not in $\langle p \rangle$.

Conversely, we prove that if $c = 1$ then $\mathsf{sat}(p) \subseteq \langle p \rangle$. For any $q \in \mathsf{sat}(p)$, there exist $n \in \mathbb{Z}_{\geq 0}$ and $\beta \in \mathbf{k}[\boldsymbol{y}][x]$ such that $a_e^n q = \beta p$. Taking the content w.r.t. x, we have

$$\begin{aligned} a_e^n \mathsf{content}(q, x) &= \mathsf{content}(\beta, x)\, \mathsf{content}(p, x) \\ &= \mathsf{content}(\beta, x) \end{aligned}$$

Thus $a_e^n \mid \beta$. There exists $\beta' \in \mathbf{k}[\boldsymbol{y}][x]$ such that $\beta = a_e^n \beta'$. So we have $a_e^n q = \beta p = a_e^n \beta' p$, and then $q = \beta' p$, that is, $q \in \langle p \rangle$. □

Remark 3. *Let $T = \{p_1\}$ be a regular chain consisting of a single polynomial. By definition, T is primitive if and only if p_1 is weakly primitive in $R = \mathbf{k}[x_1, \ldots, x_{j-1}]$, where $x_j = \mathsf{mvar}(p_1)$. Since R is a UFD, it follows from Proposition 2, that T is primitive if and only if p_1 is primitive in the ordinary sense, that is, whenever the* gcd *of the coefficients of p_1 (as a univariate polynomial in $R[x_j]$) is 1. Therefore, the notion of primitivity for a regular chain extends that of primitivity for a polynomial.*

Theorem 4. *Let $T \subset \mathbf{k}[x_1, \ldots, x_n]$ be a regular chain. Then T is primitive if and only if $\langle T \rangle = \mathsf{sat}(T)$.*

Proof. We prove the theorem by induction on the number of polynomials in T. The base case is Proposition 4, where $|T| = 1$. Now assume that $T = \{p_1, \ldots, p_m\}$ consists of $m \geq 2$ polynomials with $\mathsf{mvar}(p_1) \prec \cdots \prec \mathsf{mvar}(p_m)$. We denote by T_k the regular chain consisting of the first k polynomials in T.

First, assume indirectly that T is not primitive. We need to prove that $\langle T \rangle$ is a proper subset of $\mathsf{sat}(T)$. Let k be the smallest integer such that p_k is not weakly primitive in $R[y]$, where $y = x_j = \mathsf{mvar}(p_k)$ and $R = \mathbf{k}[x_1, \ldots, x_{j-1}]/\langle T_{k-1} \rangle$. By Proposition 4, we know $k \geq 2$.

Let $p_k = a_e y^e + \cdots + a_0$. By induction, $\mathsf{sat}(T_{k-1}) = \langle T_{k-1} \rangle$ holds and thus a_e is regular in R. Since p_k is not weakly primitive over R, there exists $\beta \in \mathbf{k}[x_1, \ldots, x_{j-1}]$ such that, in R, we have

$$(\forall 0 \leq r \leq e-1) \;\; a_e \mid \beta a_r \quad \text{and} \quad a_e \nmid \beta.$$

Define $q_k = \beta p_k / a_e$. Then $q_k \in R[y]$, since

$$\frac{\beta}{a_e} p_k = \beta y^e + \sum_{0 \leq r < e} \frac{\beta a_r}{a_e} y^r.$$

We claim that $q_k \in \langle p_k \rangle : a_e^\infty$ and $q_k \notin \langle p_k \rangle$ in $R[y]$, which leads to $\mathsf{sat}(T_k) \neq \langle T_k \rangle$.

Indeed, we have $a_e q_k = \beta p_k \in \langle p_k \rangle$ in $R[y]$. Thus, $q_k \in \langle p_k \rangle : a_e^\infty$. Now if $q_k \in \langle p_k \rangle$, there exists $\alpha \in R[y]$ such that $q_k = \alpha p_k$ in $R[y]$. By the construction of q_k, $\mathsf{deg}(q_k, y)$ equals $\mathsf{deg}(p_k, y)$. Hence $\alpha \in R$ and $\beta - \alpha a_e = 0$ in R. This contradicts $a_e \nmid \beta$.

Secondly, we assume that T is primitive and show $\langle T \rangle = \mathsf{sat}(T)$. By induction, $\mathsf{sat}(T_{k-1}) = \langle T_{k-1} \rangle$ holds. We shall prove that $\mathsf{sat}(T_k) = \langle T_k \rangle$ holds, too. To do so, we consider $p \in \mathsf{sat}(T_k)$ and show that we have $p \in \langle T_k \rangle$. Let $\mathsf{mvar}(p) = x_i$ and $\mathsf{mvar}(p_k) = x_j$. If $i > j$, then $p \in \mathsf{sat}(T_k)$ if and only if all coefficients of p w.r.t x_i are in $\mathsf{sat}(T_k)$, since T_k is a regular chain. So we can concentrate on the case $p \in \mathbf{k}[x_1, \ldots, x_j]$.

Let h_{p_k} be the leading coefficient of p_k w.r.t. $y = x_j$, that is, w.r.t. the main variable of p_k. By virtue of Theorem 3 we have

$$\begin{aligned} \mathsf{sat}(T_k) &= \langle \mathsf{sat}(T_{k-1}), p_k \rangle : h_{p_k}^\infty \\ &= \langle \langle T_{k-1} \rangle, p_k \rangle : h_{p_k}^\infty. \end{aligned}$$

By virtue of Theorem 2 we have $\mathsf{prem}(p, T_k) = 0$, since $p \in \mathsf{sat}(T_k)$. Consequently, $\mathsf{prem}(p, p_k)$ is in $\mathsf{sat}(T_{k-1}) = \langle T_{k-1} \rangle$. Now the pseudo-division formula (2.1) in Sec-

tion 2.2 leads to

$$h_{p_k}^{\alpha} p = \mathsf{pquo}(p, p_k) p_k + \mathsf{prem}(p, p_k), \tag{2.14}$$

where $\alpha = \max\{0,\ \mathsf{deg}(p,\, y) - \mathsf{deg}(p_k,\, y) + 1\}$. If $\mathsf{deg}(p,\, y) < \mathsf{deg}(p_k,\, y)$, then $p = \mathsf{prem}(p, p_k) \in \langle T_{k-1} \rangle \subset \langle T_k \rangle$ holds and we are done. From now on, we assume $\mathsf{deg}(p,\, y) \geq \mathsf{deg}(p_k,\, y)$ and we write $\alpha = \mathsf{deg}(p,\, y) - \mathsf{deg}(p_k,\, y) + 1$. With (2.14) we observe that we have the following equation in $R[y]$

$$h_{p_k}^{\alpha}\, p = q\, p_k. \tag{2.15}$$

We consider a more general situation: let $s \in \mathsf{sat}(T_k)$, let δ be a non-negative integer and let $u \in \mathbf{k}[x_1, \ldots, x_n]$ such that

$$h_{p_k}^{\delta} s = u\, p_k \tag{2.16}$$

holds in $R[y]$. In order to prove that $p \in \langle T_k \rangle$ holds, we prove that $s \in \langle T_k \rangle$ by induction on the number of terms in u. For simplicity, we denote

$$p_k = \sum_{i=0}^{e} a_i y^i \quad \text{and} \quad u = \sum_{i=0}^{f} b_i y^i,$$

with $a_e \neq 0$ and $b_f \neq 0$. Note that $a_e = h_{p_k}$.

If $u = 0$ in $R[y]$, then ${a_e}^{\delta} s = 0$ in $R[y]$. Since a_e is regular in R, we deduce $s = 0$ in $R[y]$, that is, $s \in \langle T_{k-1} \rangle$ and thus $s \in \langle T_k \rangle$. Assume $u \neq 0$ in $R[y]$. Let f' be the largest integer such that $b_{f'} \notin \langle T_{k-1} \rangle$ and write $u' = \sum_{i=0}^{f'} b_i y^i$. We have

$$a_e^{\delta} s = u' p_k \text{ in } R[y]. \tag{2.17}$$

By Lemma 1, for any $0 \leq i \leq e$, we have $a_e^{\delta} \mid b_{f'} a_i^{f'+1}$ in R. Since p_k is weakly primitive in $R[y]$, by Proposition 3 we have $a_e^{\delta} \mid b_{f'}$ in R. Thus there exists $\gamma \in \mathbf{k}[x_1, \ldots, x_{j-1}]$, $\gamma \neq 0$ in R, such that

$$a_e^{\delta} \gamma = b_{f'} \quad \text{in } R. \tag{2.18}$$

We define

$$s' = s - \gamma y^{f'} p_k. \tag{2.19}$$

Since $s \in \mathsf{sat}(T_k)$ we have $s' \in \mathsf{sat}(T_k)$. Moreover we have

$$u' = a_e^{\delta} \gamma y^{f'} + \mathsf{tail}(u').$$

Therefore, the following holds in $R[y]$:

$$a_e^{\delta} s' = \mathsf{tail}(u') p_k. \tag{2.20}$$

By induction hypothesis we have $s' \in \langle T_k \rangle$. With (2.19) we conclude $s \in \langle T_k \rangle$, as desired. □

2.5 Weak primitivity test

In this section, we point out the component-wise nature of weak primitivity. That is, if R can be written as a direct product of rings, then checking weak primitivity over R reduces to checking weak primitivity over each of its "components".

Lemma 4. *Let $R_1, \ldots, R_n$ be commutative rings with 1. Let $R = \Pi_{i=1}^{n} R_i$ be their direct product and let π_k be the canonical projection from R to R_k. Let $a, b \in R$. Then $a \mid b$ in R if and only if $\pi_k(a) \mid \pi_k(b)$ for each $1 \leq k \leq n$.*

The proof of this lemma is straightforward, and thus is omitted.

Proof. It is clear that $a \mid b$ implies $\pi_k(a) \mid \pi_k(b)$ for all k. On the another hand, assume that $\pi_k(a) \mid \pi_k(b)$ for each $1 \leq k \leq n$. Then there exists $u_k \in R_k$ such that $\pi_k(a)u_k = \pi_k(b)$. Define $u = (u_1, \ldots, u_n) \in R$. Then $\pi_k(u) = u_k$. Hence $\pi_k(a)\pi_k(u) = \pi_k(b)$, that is, $au - b \in \ker(\pi_k)$. So $au - b \in \cap_{k=1}^{n} \ker(\pi_k) = \langle 0 \rangle$. We have $a \mid b$ in R. □

Proposition 5. *Let $R = \Pi_{i=1}^{n} R_i$ be a direct product of rings and let π_k be the canonical projection from R to R_k and τ_k be the canonical injection from R_k to R. Let $p = \sum_{i=0}^{e} a_i x^i \in R[x]$ be a polynomial with a_e being regular in R. Then p is weakly primitive in $R[x]$ if and only if $\pi_k(p) = \sum_{i=1}^{e} \pi_k(a_i) x^i$ is weakly primitive in $R_k[x]$ for each $1 \leq k \leq n$.*

Proof. For any $1 \leq k \leq n$, denote $p_k = \pi_k(p)$. Since a_e is regular in R, $\pi_k(a_e) \neq 0$ for each k, and then each p_k is a polynomial of degree e.

First we prove that if all p_k are weakly primitive then p is also weakly primitive. Let $\beta \in R$ satisfying $a_e \mid a_i \beta$ for $0 \leq i \leq e-1$. By definition, we need to prove that $a_e \mid \beta$ in R.

Applying π_k to $a_e \mid a_i\beta$, we have $\pi_k(a_e) \mid \pi_k(a_i)\pi_k(\beta)$, for $0 \leq i \leq e-1$. By the weak primitivity of p_k, we have $\pi_k(a_e) \mid \pi_k(\beta)$. So there exists $u_k \in R_k$ such that $\pi_k(a_e)u_k = \pi_k(\beta)$. Define $u = (u_1, \ldots, u_n) \in \Pi_{i=1}^n R_i$. Then $\pi_k(u) = u_k$, and hence $\pi_k(a_e)\pi_k(u) = \pi_k(\beta)$, for each $1 \leq k \leq n$. By Lemma 4, $a_e \mid \beta$ in R. We proved that p is weakly primitive in $R[x]$.

On the other hand, we prove that, if p_k is not weakly primitive over R_k for *some* $1 \leq k \leq n$ then p is not weakly primitive over R. For simplicity, we assume $k = 1$. So, there exists $\beta_1 \in R_1$ such that $\pi_1(a_e) \mid \pi_1(a_i)\beta_1$ for $0 \leq i \leq e-1$, but $\pi_1(a_e) \nmid \beta_1$. Define $\beta = \tau_1(\beta_1) = (\beta_1, 0, \ldots, 0) \in R$. Then we claim that $a_e \nmid \beta$ and $a_e \mid a_i\beta$ for $0 \leq i \leq e-1$. This implies that p is not weakly primitive over R, as desired.

Indeed, first we have $a_e \nmid \beta$, since $\pi_1(a_e) \nmid \pi_1(\beta) = \beta_1$. Second, to prove $a_e \mid a_i\beta$ for $0 \leq i \leq e-1$, by Lemma 4, we need to prove that $\pi_k(a_e) \mid \pi_k(a_i\beta)$ for $1 \leq k \leq n$ and $0 \leq i \leq e-1$. If $k = 1$, it follows from the choice of β_1. If $2 \leq k \leq n$, we have

$$\pi_k(a_i\beta) = \pi_k(a_i)\pi_k(\beta) = \pi_k(a_i) \cdot 0 = 0$$

for $1 \leq i \leq e-1$. Thus $\pi_k(a_e) \mid \pi_k(a_i\beta)$ holds for $1 \leq i \leq e-1$. □

Example 3. *Let $T = \{p_1, p_2\}$ be a regular chain in $\mathbb{Q}[t \prec x \prec y]$ with $p_1 = x(x-t), p_2 = (x+t)y + t$. Since $p_1 = x^2 - tx$ is strongly primitive in $(\mathbb{Q}[t])[x]$, p_1 is weakly primitive in $(\mathbb{Q}[t])[x]$. Let $R = \mathbb{Q}[t, x]/\langle x(x-t)\rangle$. Then we have*

$$R = R_1 \times R_2 = \mathbb{Q}[x, t]/\langle x\rangle \times \mathbb{Q}[x, t]/\langle x - t\rangle \simeq \mathbb{Q}[t] \times \mathbb{Q}[t].$$

Over R_1, $p_2 = ty + t$ is not weakly primitive, since t is not invertible over R_1 and according to the definition we can choose $\beta = 1$. Hence T is not a primitive regular chain.

In order to generalize the construction of the above example into an algorithm, one would need to use algebraic factorization. In the next section, we propose a primitivity test for regular chains which avoids algebraic factorization, relying instead on polynomial GCDs modulo regular chains. Based on the algorithms and software tools available today we view it as a practical solution, as confirmed in Section 2.8.

2.6 A primitivity test algorithm

In Section 2.4, we define the notion of primitive regular chain which generalizes that of primitive polynomial over a UFD. In this section, we present another characterization

on primitivity in terms of regularity of a polynomial. As a consequence, we obtain an algorithm to test whether a regular chain is primitive or not.

Lemma 5, 6, 7 and 8 are well-known facts. The proofs of Lemma 5 and Lemma 8 are straightforward. Lemma 6 can be found as Lemma 9.2.3 in [40] whereas Lemma 7 is in [80], Lemma 7.

Lemma 5. *Let I be a proper ideal of R and let h be an element of R. Then h is regular modulo I if and only if $I = I : h^\infty$ holds.*

Proof. Assume that h is regular modulo I. For any $f \in I : h^\infty$, there exists a nonnegative integer m such that $fh^m \in I$. Since h is regular modulo I, h^m is also regular modulo I. So $f \in I$ holds.

Conversely, assume that $I = I : h^\infty$ holds. Since I is a proper ideal, h cannot belong to I. If h is not regular modulo I, then there exists an element $g \in R \setminus I$ such that $gh \in I$. Hence $g \in I : h^\infty = I$ holds, a contradiction. □

Lemma 6. *Let a and b be two regular elements of R. Assume that a and b are not invertible. If a is regular modulo $\langle b \rangle$ then b is also regular modulo $\langle a \rangle$.*

Proof. Observe that, since a and b are not invertible, both $\langle a \rangle$ and $\langle b \rangle$ are proper. First, we prove that b is not in $\langle a \rangle$. Suppose b is in $\langle a \rangle$. There exists an $x \in R$ such that $b = xa$. Since a is regular modulo $\langle b \rangle$, x is in $\langle b \rangle$, that is, there exists an $x' \in R$ such that $x = bx'$. It follows that $b = bax'$ holds. Since b is a regular element of R, $ax' = 1$ holds in R, which contradicts to the fact that a is not invertible. Now according to the definition, we only need to show that $x \in \langle a \rangle$ holds, for any $x \in R$ satisfying $bx \in \langle a \rangle$. Indeed, there exists $x' \in R$ such that $bx = ax'$. Since a is regular modulo $\langle b \rangle$, we have $x' \in \langle b \rangle$, that is, there exists $x'' \in R$ such that $x' = bx''$. So $bx = ax' = bax''$ holds. Since b is a regular element of R, $x = ax'' \in R$ holds, as desired. □

Lemma 7 (Mc Coy Theorem)**.** *A non-zero polynomial $f \in R[x]$ is a zerodivisor if and only if there exists a non-zero element $a \in R$ such that $af = 0$ holds.*

Lemma 8. *Let $f \in R[x]$ be a non-constant polynomial. If its leading coefficient is a regular element in R, then f is not a unit.*

Proposition 6. *Let R be a Noetherian commutative ring with 1. Consider a polynomial $f = \sum_{i=0}^{n} a_i x^i \in R[x]$. Assume that n is at least 1 and a_n is regular in R. Then $\langle f \rangle = \langle f \rangle : a_n^\infty$ holds if and only if a_n is invertible in R, or $\mathsf{tail}(f)$ is regular modulo $\langle a_n \rangle$.*

Proof. If a_n is invertible in R, then clearly $\langle f\rangle : a_n^\infty = \langle f\rangle$ holds. So we assume that a_n is not invertible in R. Note that both a_n and f are regular in $R[x]$; this follows from Lemma 7. Since a_n is not invertible in R, a_n is not invertible in $R[x]$ either. Since a_n is regular in R, it follows from Lemma 8 that f is not invertible in $R[x]$. Then, applying Lemma 5 and 6, we deduce

$$\begin{aligned}\langle f\rangle = \langle f\rangle : {a_n}^\infty &\iff a_n \text{ is regular modulo } \langle f\rangle\\ &\iff f \text{ is regular modulo } \langle a_n\rangle\\ &\iff \mathsf{tail}(f) \text{ is regular modulo } \langle a_n\rangle.\end{aligned}$$

This completes the proof. □

The following corollary may be seen as another characterization of the primitivity of a regular chain. This also provides an algorithm for checking whether a regular chain is primitive or not.

Corollary 1 (Primitivity test of a regular chain)**.** *Let $T \subset \mathbf{k}[x_1, \ldots, x_{s-1}]$ be a primitive regular chain. Let $p = \sum_{i=0}^{e} a_i x_s^i \in \mathbf{k}[x_1, \ldots, x_s]$ with a_e being regular modulo $\mathsf{sat}(T)$. Denote $\mathsf{tail}(p) = \sum_{i=0}^{e-1} a_i x_s^i$. Then $T \cup \{p\}$ is a primitive regular chain if and only if a_e is invertible modulo $\mathsf{sat}(T)$, or $\mathsf{tail}(p)$ is a regular polynomial modulo $\langle T \cup \{a_e\}\rangle$.*

Proof. This is a direct consequence of Proposition 6, Theorem 4 and the definition of a regular chain. □

Thus the problem of checking whether a regular chain $T \cup \{p\}$ is primitive or not, reduces to checking whether the polynomial $\mathsf{tail}(p)$ is regular or not modulo $\langle T \cup \{a_e\}\rangle$. We next show that (T, a_e) in Corollary 1 generates an unmixed ideal; this result is crucial in view of Algorithm 1 below. Indeed, it allows us to deal with the following subtle point: a polynomial p regular modulo the radical $\sqrt{I}$ of an ideal I may not be regular modulo I. For example, consider $p = y$ and $I = \langle xy, x^2\rangle$. Then y is a zerodivisor modulo I but y is regular modulo $\sqrt{I} = \langle x\rangle$. If I is unmixed, then p is regular modulo I if and only if p is regular modulo $\sqrt{I}$.

Lemma 9. *Let $R = \mathbf{k}[x_1, \ldots, x_n]$ and T be a primitive regular chain of R. If $t \in R$ is regular but not invertible modulo $\mathsf{sat}(T)$, then (T, t) is a regular sequence of R and the ideal $\langle T, t\rangle$ is unmixed with dimension $n - |T| - 1$.*

Proof. Denote $T_i = T \cap \mathbf{k}[x_1, \ldots, x_i]$. Since T is primitive, $\mathsf{sat}(T_i) = \langle T_i \rangle$ holds for each i. Thus T is already a regular sequence of R. Now since t is regular but not invertible modulo $\mathsf{sat}(T) = \langle T \rangle$, by definition (T, t) is a regular sequence.

Let $I = \langle T, t \rangle$ and $d = |T|$. According to the Principal Ideal Theorem [see 29, Theorem 10.2] the dimension $\dim(I)$ of I is at least $n - (d + 1)$. On the other hand, since (T, t) is a regular sequence of length $d + 1$, the dimension of I is at most $n - (d + 1)$. Hence, $\dim(I) = n - (d + 1)$ and then I is unmixed, by Macaulay Unmixedness Theorem [see 79, Theorem 5.7]. □

Algorithm 1: IsPrimitive(T)

Input : T, a regular chain of $\mathbf{k}[x_1, \ldots, x_n]$
Output : true if T is primitive, false otherwise

```
1  if |T| = 1 then
2      t ← the defining polynomial of T
3      if content(t, mvar(t)) ∈ k then
4          return true
5      else return false
6  else
7      write T as T' ∪ {t}, where t has the greatest main variable
8      if not IsPrimitive(T') then
9          return false
10     else
11         h ← init(t), r ← tail(t)
12         for U ∈ Triangularize(T' ∪ {h}) do
13             if iterRes(r, U) = 0 then return false
14         return true
```

Remark 4. *Before proving the correctness of the above algorithm, we comment on its subprocedures and possible optimization.*

(1) *The function* Triangularize *decomposes a polynomial system F into a finite set of regular chains U_i such that $\sqrt{\langle F \rangle} = \cap_i \sqrt{\mathsf{sat}(U_i)}$ holds; this is called a triangular decomposition of F in the sense of Kalkbrener [7]. According to the above specification, the set of the associated primes of $\sqrt{\langle F \rangle}$ are "implicitly" represented by U_i's .*

Triangularize *is one of the core functions in the* RegularChains *library in* Maple *[48]; it implements the triangular decomposition algorithm of [65].*

While computing in Kalkbrener's sense, it has the same specification as the function $\mathbf{solve}_n$ *in [41], although the algorithms of [64] and [41] are quite different.*

Apart from Kalkbrener's sense, Triangularize *can also work in the Lazard sense [see 7], where all solutions of the input systems will be explicitly represented by means of regular chains. In general, this function runs faster in Kalkbrener's sense, since only* generic *solutions will be represented explicitly.*

(2) *The use of* Triangularize *seems hard to avoid. The purpose is to represent all associated primes of the ideal* $\langle T \cup \{h\}\rangle$ *by means of regular chains. Geometrically, it is the intersection of the zero set of* T *with the hypersurface defined by* h*.*

(3) *Algorithm 1 can be optimized using Item* (1) *of Remark 2: if a coefficient* a_i *of* $t = a_e x^e + \cdots + a_0$ *is an invertible constant, then lines 11-13 can be skipped since* t *is strongly primitive.*

Proof. We prove the above algorithm IsPrimitive. Termination of the algorithm follows from the fact that in each recursive call the number of polynomials in the input regular chain decreases by 1.

For the correctness, we proceed by induction on the number of polynomials in the regular chain T. When $|T| = 1$, correctness follows from Remark 3. So we assume $|T| > 1$. Definition 4 and Theorem 4 imply that if T is primitive then T' is also primitive. So we assume that T' is primitive and branch to line 10.

Let $\mathcal{U}$ be the output of Triangularize in line 10 and let $I = \langle T' \cup \{h\}\rangle$. From the specification of Triangularize, we have

$$\bigcap_{U \in \mathcal{U}} \sqrt{\mathsf{sat}(U)} = \sqrt{I}.$$

By Corollary 1, we need to distinguish two cases: h is invertible (resp. not invertible) modulo $\langle T'\rangle = \mathsf{sat}(T')$.

If h is invertible modulo $\langle T'\rangle$ then $\mathcal{U}$ is empty, and the algorithm correctly returns true. Assume from now on that h is not invertible modulo $\langle T'\rangle$. In this case by Lemma 9, the triangular decomposition $\mathcal{U}$ is not empty. So T is primitive if and only if r is regular modulo I. By Lemma 9 again, the ideal I is unmixed and therefore T is primitive if and only if r is regular modulo $\sqrt{I}$. This holds if and only if r is regular modulo $\mathsf{sat}(U)$ for each $U \in \mathcal{U}$. Finally, the correctness of Algorithm 1 follows from Theorem 2. □

Example 4. *Let $R = \mathbf{k}[z \prec y \prec x]$ be a polynomial ring and $T = \{t_1, t_2\}$ be a regular chain of R with $t_1 = y^5 - z^4, t_2 = zx - y^2$. Clearly, $\{t_1\}$ is a primitive regular chain. Let $I = \langle t_1, \mathsf{lc}(t_2)\rangle = \langle t_1, z\rangle = \langle z, y^5\rangle$. In Algorithm 1 the call to* Triangularize *will produce $\sqrt{I} = \sqrt{\mathsf{sat}(U)}$ where $U = \{z, y\}$ is a regular chain. Thus, the computation*

$$\mathsf{iterRes}(\mathsf{tail}(t_2), U) = \mathsf{iterRes}(-y^2, U) = 0$$

implies that $\mathsf{tail}(t_2) = -y^2$ is not regular modulo I. Thus T is not primitive. In fact, the prime ideal $\mathsf{sat}(T) = \langle x^3 - yz, xz - y^2, z^2 - x^2y\rangle$ can not be generated by only two polynomials [see 21, page 43]. Hence, in any *variable ordering, one cannot find a primitive regular chain C such that $\langle C\rangle = \mathsf{sat}(T)$.*

2.7 An application to inclusion test

A fundamental problem in the theory of regular chains is the inclusion test for saturated ideals, that is, deciding if $\mathsf{sat}(T) \subseteq \mathsf{sat}(U)$ holds for two regular chains T and U. For a regular chain T, denote by $\mathsf{mvar}(T)$ the set of main variables of polynomials in T, which is also called the set of algebraic variables of T. In this section, we first show that when T and U share the same set of algebraic variables the inclusion test is simple. Then we point out that the notion of primitivity presented in this chapter solves the inclusion test problem partially.

Lemma 10. *Let T and U be two regular chains. If $\mathsf{sat}(T) \subseteq \mathsf{sat}(U)$ and $|T| = |U|$ hold, then each associated prime of $\mathsf{sat}(U)$ is also an associated prime of $\mathsf{sat}(T)$.*

Proof. Let $\mathcal{T}$ and $\mathcal{U}$ be the set of associated primes of $\mathsf{sat}(T)$ and $\mathsf{sat}(U)$ respectively. Then we have

$$\sqrt{\mathsf{sat}(T)} = \bigcap_{P \in \mathcal{T}} P \quad \text{and} \quad \sqrt{\mathsf{sat}(U)} = \bigcap_{Q \in \mathcal{U}} Q.$$

Since $\mathsf{sat}(T) \subseteq \mathsf{sat}(U)$ implies $\sqrt{\mathsf{sat}(T)} \subseteq \sqrt{\mathsf{sat}(U)}$, for each $Q \in \mathcal{U}$ there exists $P \in \mathcal{T}$ such that $P \subseteq Q$. Since T and U are unmixed with same height, $\dim(P)$ equals $\dim(Q)$, which implies $Q = P$. Hence $\mathcal{U}$ is a subset of $\mathcal{T}$. □

Proposition 7. *Let T and U be two regular chains with the same set of algebraic variables. Write T as $T = T' \cup \{t\}$ with t having largest main variable. Then $\mathsf{sat}(T) \subseteq \mathsf{sat}(U)$ if and only if $\mathsf{sat}(T') \subseteq \mathsf{sat}(U)$ and $\mathsf{prem}(t, U) = 0$.*

Proof. Clearly, we only need to show that $\mathsf{sat}(T) \subseteq \mathsf{sat}(U)$ holds if $\mathsf{sat}(T') \subseteq \mathsf{sat}(U)$ and $\mathsf{prem}(t, U) = 0$.

Denote by h the initial of t. We first prove that h is regular modulo $\mathsf{sat}(U)$. Since h is regular modulo $\mathsf{sat}(T')$, h is not contained in any associated prime of $\mathsf{sat}(T')$. Let u be the polynomial in U such that $\mathsf{mvar}(t) = \mathsf{mvar}(u)$ and define $U' = U \setminus \{u\}$. Then we have $\mathsf{sat}(T') \subseteq \mathsf{sat}(U')$. By Lemma 10, h is not contained in any associated prime of $\mathsf{sat}(U')$. Hence h is regular modulo $\mathsf{sat}(U')$. It follows that h is regular modulo $\mathsf{sat}(U)$ since the main variable of h is smaller than that of u.

For arbitrary $f \in \mathsf{sat}(T)$, we have $\mathsf{prem}(f, t) \in \mathsf{sat}(T') \subseteq \mathsf{sat}(U)$. By the pseudo-division formula, $h^e f = \mathsf{prem}(f, t) + q\, t$ for some $e \geq 0$ and some q. Since $\mathsf{prem}(t, U) = 0$, we have $t \in \mathsf{sat}(U)$. Therefore $h^e f$ belongs to $\mathsf{sat}(U)$, which implies $f \in \mathsf{sat}(U)$ since h is regular modulo $\mathsf{sat}(U)$. □

The above proposition handles the case in which two regular chains have the same set of algebraic variables.

Example 5. *Let* $R = \mathbf{k}[x \prec y \prec z]$ *and let* $T = \{xz + y\}$ *and* $U = \{x, y\}$ *be regular chains of* R. *Then* $\mathsf{sat}(T) = \langle xz + y\rangle \subsetneq \langle x, y\rangle = \mathsf{sat}(U)$ *holds, although we have* $\mathsf{mvar}(T) = \{z\}$ *and* $\mathsf{mvar}(U) = \{x, y\}$.

In practice, the inclusion $\mathsf{sat}(T) \subseteq \mathsf{sat}(U)$ is often established by proving that $\langle T\rangle \subseteq \mathsf{sat}(U)$ holds and that all initials in T are regular modulo $\mathsf{sat}(U)$. This simple criterion follows immediately from the definition of a saturated ideal and Lemma 5.

Now with the notion of primitivity for a regular chain, we have another useful way to detect if $\mathsf{sat}(T) \subseteq \mathsf{sat}(U)$ holds. That is, $\mathsf{sat}(T) \subseteq \mathsf{sat}(U)$ holds whenever $\langle T\rangle \subseteq \mathsf{sat}(U)$ holds and T is primitive. In the above example, the initial of $zx + y$ is not regular modulo $\mathsf{sat}(U)$. However, we know that $\mathsf{sat}(T)$ is contained in $\mathsf{sat}(U)$, since T is primitive and $\langle T\rangle \subseteq \mathsf{sat}(U)$ holds. In the following Section 2.8, we shall see that algorithm IsPrimitive is efficient and primitive regular chains appear quite often in practice.

Corollary 2 below is a direct consequence of Proposition 7, which shows that it is an easy task to check whether two regular chains have the same saturated ideal. Actually, testing $\mathsf{sat}(T) = \mathsf{sat}(U)$ can be done "directly" without testing the inclusions $\mathsf{sat}(T) \subseteq \mathsf{sat}(U)$ and $\mathsf{sat}(U) \subseteq \mathsf{sat}(T)$. The algorithm concluding this section combines together the different criteria reported above for testing the inclusion of saturated ideals. Observe that this algorithm is not always able to check whether the inclusion holds or not.

Corollary 2. *Let* $T = T' \cup \{t\}$ *and* $U = U' \cup \{u\}$ *be two regular chains with* t *and* u *having the greatest main variable in* T *and* U *respectively. The equality* $\mathsf{sat}(T) = \mathsf{sat}(U)$ *holds if and only if the following conditions hold*

1. $\mathsf{sat}(T') = \mathsf{sat}(U')$,

2. $\mathsf{mvar}(t) = \mathsf{mvar}(u)$,

3. $t \in \mathsf{sat}(U)$ *and* $u \in \mathsf{sat}(T)$.

Algorithm 2: IsIncluded(T, U)

Input : T and U, regular chains of $\mathbf{k}[x_1, \ldots, x_n]$
Output : If **true** (resp. **false**) is returned then $\mathsf{sat}(T) \subseteq \mathsf{sat}(U)$ holds (resp. does not hold). If **fail** is returned then the inclusion could not be proved nor disproved.

```
if T = ∅ then return true
if U = ∅ then return false
if mvar(T) = mvar(U) then
    v ← max(mvar(T)), T' ← T \ {T_v}
    if IsIncluded(T', U) and prem(T_v, U) = 0 then return true
if T ⊆ sat(U) then
    if iterRes(∏_{t∈T} init(t), U) ≠ 0 then return true
    if IsPrimitive(T) then return true
return fail
```

2.8 Experimentation

We have implemented algorithm IsPrimitive on top of the RegularChains library in Maple [48]. The experimentation, described hereafter, was conducted on well-known problems used in [14][1], and the tests were performed in Maple 11 on an Intel Pentium 4 machine (3.20GHz CPU, 2.0GB memory).

First, we computed triangular decompositions using the Triangularize command in the sense of Kalkbrener. Then, we applied the IsPrimitive algorithm to each regular chain in the output.

In Table 2.1, we summarize the features of the problems and our experimental results. The name of the problems are listed in the first column. The second column gives the number n of variables and the maximal total degree d. For each triangular decomposition (which is a list of regular chains) we record the total running time (in seconds) of IsPrimitive in the third column. The last column is the result of mapping

[1]The defining polynomial systems can be found at `http://www.orcca.on.ca/~panwei/issac08/`

IsPrimitive to each triangular decomposition: in each of these patterns Y stands for *true* and N for *false*.

These data show that the procedure IsPrimitive is efficient in practice. This agrees with the fact that, in Algorithm 1, the input polynomial set in each call to Triangularize is rather structured. We also observe that primitive regular chains appear quite often in the output of triangular decompositions.

Table 2.1: Tests for IsPrimitive on 14 examples

System	(n, d)	Time	Pattern
KdV575	(26, 3)	3.525	[Y, Y, Y, Y, Y, Y, Y]
MontesS11	(6, 4)	.001	[Y]
MontesS16	(15, 2)	.103	[Y, Y, Y, N, Y, Y, Y]
Wu-Wang2	(13, 3)	0.099	[Y, N, Y, Y, Y]
MontesS10	(7, 3)	.145	[N]
Lazard2001	(7, 4)	2.314	[Y, Y, Y, N, Y, N]
Lanconelli	(11, 3)	.062	[N, Y]
Wang93	(5, 3)	.142	[N]
Leykin-1	(8, 4)	.228	[Y, Y, Y, Y, Y, Y, Y, Y, N, Y, Y, Y, N, N]
MontesS14	(5, 4)	1.171	[Y, N, N]
MontesS15	(12, 2)	.312	[N]
Maclane	(10, 2)	.157	[Y, Y, N, Y, N]
MontesS12	(8, 2)	.042	[N]
Liu-Lorenz	(5, 2)	1.117	[N, Y]

2.9 Discussion

We have generalized the notion of primitivity from univariate polynomials to regular chains. This has allowed us to establish a necessary and sufficient condition for a regular chain T to generate its saturated ideal $\mathsf{sat}(T)$. Assume that T is not empty and write $T = T' \cup \{p\}$ where p is the polynomial of T with largest main variable. Theorem 4 states that the equality $\langle T \rangle = \mathsf{sat}(T)$ holds whenever $\langle T' \rangle = \mathsf{sat}(T')$ holds and the polynomial p is *weakly primitive* over $\mathbf{k}[\boldsymbol{x}]/\langle T' \rangle$. This latter property is a generalization of the usual notion of primitivity for polynomials over a UFD.

Examining the proof of Theorem 4, we make the following observation. When p is not weakly primitive over $\mathbf{k}[\boldsymbol{x}]/\langle T' \rangle$, the proof exhibits a polynomial q which belongs to $\mathsf{sat}(T)$ but not to $\langle T \rangle$. When p is weakly primitive over $\mathbf{k}[\boldsymbol{x}]/\langle T' \rangle$, the proof shows that every polynomial q of $\mathsf{sat}(T)$ belongs to $\langle T \rangle$. The argument is constructive

providing that one has at hand an algorithm for dividing a by b modulo $\langle T' \rangle$, where b is a polynomial regular modulo $\langle T' \rangle$ and is a multiple of the polynomial a modulo $\langle T' \rangle$. This can be done via Gröbner basis computations [see 60]. An algorithmic solution based on the algorithms of the RegularChains library is an ongoing research work.

Theorem 4 and its proof do not lead directly to an algorithm for testing the equality $\langle T \rangle = \mathsf{sat}(T)$. Algorithm 1 provides such a decision procedure. This algorithm reduces to testing whether a polynomial is regular modulo an ideal. Fortunately the involved ideal is unmixed which allows us to rely on the algorithms of the RegularChains library avoiding Gröbner basis computations. Our experimentation illustrates the practical efficiency of Algorithm 1.

Algorithm 1 does not generalize easily in the differential setting. Indeed, consider the polynomial $p = u_x^2 - 4u$ as in [74, example 1 page 120]. We recall hereafter that we have $[u_x^2 - 4u] \subsetneq [u_x^2 - 4u] : \{u_x\}^\infty$. This indicates that even in the case of a single polynomial, the problem is much harder in the differential setting since the case of a single polynomial in the algebraic setting is obvious (line 3 of Algorithm 1). It is obvious to show that $u_{xx} - 2 \in [u_x^2 - 4u] : \{u_x\}^\infty$ since $dp/dx = 2u_x(u_{xx} - 2)$. However $u_{xx} - 2 \notin [u_x^2 - 4u]$ holds for the following reason: the solution $u = 0$ for $[u_x^2 - 4u]$ does not cancel $u_{xx} - 2$ which implies: $u_{xx} - 2 \notin [u_x^2 - 4u]$. Thus, we have; $[u_x^2 - 4u] \subsetneq [u_x^2 - 4u] : \{u_x\}^\infty$. In [6] the authors have shown that any lexicographical reduced Gröbner basis of a prime ideal $\mathcal{P}$ contains a regular chain T such that $\mathsf{sat}(T) = \mathcal{P}$ holds. One would like to be able to reverse this construction, that is, retrieving from T a Gröbner basis of $\mathcal{P}$. Kalkbrener's formula (Property (3) in Theorem 3) and the notion of primitive regular chains seem to form a good starting point for investigating this question.

In general a primitive regular chain is *not* a lex Gröbner basis with the same variable ordering. Over the ring $\mathbb{Q}[x \succ y \succ z \succ v \succ u \succ r \succ t]$, the following regular chain T is primitive:

$$T = \begin{cases} v - rt, \\ ztu - 1, \\ yr - t^2u - 1, \\ xu - xr - 2\,x - u - 2\,t^4 + 1. \end{cases}$$

The ideal $\mathsf{sat}(T) = \langle T \rangle$ has the reduced lex Gröbner basis G in lex order $x \succ y \succ$

$z \succ v \succ u \succ r \succ t$, which consists of 6 polynomials.

$$G = \begin{cases} v - rt, \\ ztu - 1, \\ yr - t^2u - 1, \\ xu - xr - 2\,x - u - 2\,t^4 + 1, \\ rtzx + 1 - zt - x + 2\,ztx + 2\,t^5z, \\ ztx + t^2x - yzt - xy + y + 2\,yztx + 2\,t^5yz \end{cases}$$

Moreover, in the same variable ordering, its reduced degree reverse lex Gröbner basis G_2 consists of 12 polynomials. Clearly, when T is a primitive regular chain, no other generating set of the saturated ideal $\mathsf{sat}(T)$ can can have fewer elements than T; in addition T provides nice algorithmic properties (membership test, regularity test) as Gröbner bases do.

$$G_2 = \begin{cases} -v + rt, \\ t^2u + 1 - yr, \\ ztu - 1, \\ yr^2 - vut - r, \\ yzr - z - t, \\ xu^2 - xru - 2\,yvt - 2\,xu - u^2 + 2\,t^2 + u, \\ vzu - r, \\ yvz - zt - t^2, \\ -xu + xr + 2\,x + u + 2\,t^4 - 1, \\ 2\,vt^3 - xru + xr^2 + 2\,xr + ru - r, \\ xur^2 - xr^3 - 2\,v^2t^2 - 2\,xr^2 - r^2u + r^2, \\ xzvr^3 + 2\,zv^3t^2 + 2\,vzxr^2 - zvr^2 - xr^3 + r^3. \end{cases}$$

As discussed in Section 2.7, an application of Algorithm 1 is in the removal of redundant components for triangular decompositions in the sense of Kalkbrener. However, this Algorithm 2 provides only a criterion for removing redundant components. Obtaining a decision algorithm, free of Gröbner basis computations, for testing the inclusion of saturated ideals, remains an open problem.

Another possible research direction is to investigate the relations between primitive regular chains and the minimum number of generators of saturated ideals. For instance, it is natural to ask whether a prime ideal $\mathcal{P}$ of height h can be generated

by h elements if and only if there exists a variable ordering and a primitive regular chain C (w.r.t. this variable ordering) such that C generates $\mathcal{P}$.

Chapter 3

Regular GCD : a Bottom-up Algorithm

3.1 Introduction

Triangular decomposition of polynomial systems are based on a recursively univariate vision of multivariate polynomials. Most of the methods computing these decompositions manipulate polynomial remainder sequences (PRS). Moreover, these methods are usually "factorization free", which explains why two different irreducible components may be represented by the same regular chain. An essential routine is then to check whether a hypersurface $f = 0$ contains one of the irreducible components encoded by a regular chain T. This is achieved by testing whether the polynomial f is a zero-divisor modulo the saturated ideal of T. This univariate vision on regular chains allows to perform *regularity test* by means of GCD computations. However, since the saturated ideal of T may not be prime, the concept of a GCD used here is not standard.

The first formal definition of this type of GCDs was given by Kalkbrener in [41]. But in fact, GCDs over non-integral domains were already used in several papers [28, 35, 45] since the introduction of the celebrated *D5 Principle* [24] by Della Dora, Dicrescenzo and Duval. Indeed, this brilliant and simple observation allows one to carry out over direct product of fields computations that are usually conducted over fields. For instance, computing univariate polynomial GCDs by means of the Euclidean Algorithm.

To define a polynomial GCD of two (or more) polynomials modulo a regular chain T, Kalkbrener refers to the irreducible components that T represents. In order to

improve the practical efficiency of those GCD computations by means of subresultant techniques, Rioboo and Moreno Maza proposed a more abstract definition in [67]. Their GCD algorithm is, however, limited to regular chains with zero-dimensional saturated ideals.

While Kalkbrener's definition cover the positive dimensional case, his approach cannot support triangular decomposition methods solving polynomial systems incrementally, that is, by solving one equation after another. This is a serious limitation since incremental solving is a powerful way to develop efficient sub-algorithms, by means of geometrical consideration. The first incremental triangular decomposition method was proposed by Lazard in [44], without proof nor a GCD definition. Another such method was established by the Moreno Maza in [65] together with a formal notion of GCD adapted to the needs of incremental solving. This concept, called *regular GCD*, is reviewed in Section 3.2 in the context of regular chains. A more abstract definition follows.

Let $\mathbb{B}$ be a commutative ring with unity. Let P, Q and G be non-zero univariate polynomials in $\mathbb{B}[y]$. We say that G is a *regular GCD* of P, Q if the following three conditions hold:

(1) the leading coefficient of G in y is a regular element of $\mathbb{B}$,

(2) G lies in the ideal generated by P and Q in $\mathbb{B}[y]$, and

(3) if G has positive degree w.r.t. y, then G pseudo-divides both P and Q, that is, the pseudo-remainders $\mathsf{prem}(P, G)$ and $\mathsf{prem}(Q, G)$ are null.

In the context of regular chains, the ring $\mathbb{B}$ is the residue class ring of a polynomial ring $\mathbb{A} := \mathbf{k}[x_1, \ldots, x_n]$ over a field $\mathbf{k}$ by the saturated ideal $\mathsf{sat}(T)$ of a regular chain T. Even if the leading coefficients of P, Q are regular and $\mathsf{sat}(T)$ is radical, the polynomials P, Q may not necessarily admit a regular GCD (unless $\mathsf{sat}(T)$ is prime). However, by splitting T into several regular chains $T_1, \ldots, T_e$ (in a sense specified in Section 3.2) one can compute a regular GCD of P, Q over each of the rings $\mathbb{A}/\mathsf{sat}(T_i)$, as shown in Section 3.4.

In this and the following chapter, we propose a new algorithm for this task, together with a theoretical study and implementation report, providing dramatic improvements w.r.t. previous work [41, 65].

Section 3.3 exhibits sufficient conditions for a subresultant of P, Q (regarded as univariate polynomials in y) to be a regular GCD of P, Q w.r.t. T. Some of these properties could be known, but we could not find a reference for them, in particular when $\mathsf{sat}(T)$ is not radical.

These results reduce the computation of regular GCDs to that of subresultant chains. More precisely and reusing the above notations, Theorem 5 in Section 3.4 states that the regular chain T can be split into regular chains $T_1, \ldots, T_e$ such that for each $i = 1 \cdots e$, one of the subresultants of P and Q is a regular GCD of P, Q over $\mathbb{A}/\mathsf{sat}(T_i)$.

This chapter is based on the ISSAC 2009 article [49]. We include a formal presentation of Algorithm 3 together with a complete proof.

3.2 Preliminaries

Let $\mathbf{k}$ be a field and let $\mathbf{k}[\mathbf{x}] = \mathbf{k}[x_1, \ldots, x_n]$ be the ring of polynomials with coefficients in $\mathbf{k}$, with ordered variables $x_1 \prec \cdots \prec x_n$. Let $\overline{\mathbf{k}}$ be the algebraic closure of $\mathbf{k}$. If $\mathbf{u}$ is a subset of $\mathbf{x}$ then $\mathbf{k}(\mathbf{u})$ denotes the fraction field of $\mathbf{k}[\mathbf{u}]$. For $F \subset \mathbf{k}[\mathbf{x}]$, we denote by $\langle F \rangle$ the ideal it generates in $\mathbf{k}[\mathbf{x}]$ and by $\sqrt{\langle F \rangle}$ the radical of $\langle F \rangle$. For $H \in \mathbf{k}[\mathbf{x}]$, the *saturated ideal* of $\langle F \rangle$ w.r.t. H, denoted by $\langle F \rangle : H^\infty$, is the ideal

$$\{Q \in \mathbf{k}[\mathbf{x}] \mid \exists m \in \mathbb{N} \text{ s.t. } H^m Q \in \langle F \rangle\}.$$

A polynomial $P \in \mathbf{k}[\mathbf{x}]$ is a *zero-divisor* modulo $\langle F \rangle$ if there exists a polynomial Q such that $PQ \in \langle F \rangle$, and neither P nor Q belongs to $\langle F \rangle$. The polynomial P is *regular* modulo $\langle F \rangle$ if it is neither zero, nor a zero-divisor modulo $\langle F \rangle$. We denote by $V(F)$ the *zero set* (or algebraic variety) of F in $\overline{\mathbf{k}}^n$. For a subset $W \subset \overline{\mathbf{k}}^n$, we denote by $\overline{W}$ its closure in the Zariski topology.

3.2.1 Regular chains and related notions

Polynomial. If $P \in \mathbf{k}[\mathbf{x}]$ is a non-constant polynomial, the largest variable appearing in P is called the *main variable* of P and is denoted by $\mathsf{mvar}(P)$. We regard P as a univariate polynomial in its main variable. The degree and the leading coefficient of P as a univariate polynomial in $\mathsf{mvar}(P)$ are called *main degree* and *initial* of P; they are denoted by $\mathsf{mdeg}(P)$ and $\mathsf{init}(P)$ respectively.

Triangular Set. A subset T of non-constant polynomials of $\mathbf{k}[\mathbf{x}]$ is a *triangular set* if the polynomials in T have pairwise distinct main variables. Denote by $\mathsf{mvar}(T)$ the set of all $\mathsf{mvar}(P)$ for $P \in T$. A variable $v \in \mathbf{x}$ is *algebraic* w.r.t. T if $v \in \mathsf{mvar}(T)$; otherwise it is *free*. For a variable $v \in \mathbf{x}$ we denote by $T_{<v}$ (resp. $T_{>v}$) the subsets of T consisting of the polynomials with main variable less than (resp. greater than) v.

If $v \in \mathsf{mvar}(T)$, we denote by T_v the polynomial $P \in T$ with main variable v. For T not empty, $T_{\max}$ denotes the polynomial of T with largest main variable.

Quasi-component and saturated ideal. Given a triangular set T in $\mathbf{k}[\mathbf{x}]$, denote by h_T the product of the $\mathsf{init}(\mathrm{P})$ for all $P \in T$. The *quasi-component* $W(T)$ of T is $V(T) \setminus V(h_T)$, that is, the set of the points of $V(T)$ which do not cancel any of the initials of T. We denote by $\mathsf{sat}(T)$ the *saturated ideal* of T, defined as follows: if T is empty then $\mathsf{sat}(T)$ is the trivial ideal $\langle 0 \rangle$; otherwise it is the ideal $\langle T \rangle : h_T^\infty$.

Regular chain. A triangular set T is a *regular chain* if either T is empty, or $T \setminus \{T_{max}\}$ is a regular chain and the initial of T_{max} is regular with respect to $\mathsf{sat}(T \setminus \{T_{max}\})$. In this latter case, $\mathsf{sat}(T)$ is a proper ideal of $\mathbf{k}[\mathbf{x}]$. From now on $T \subset \mathbf{k}[\mathbf{x}]$ is a regular chain; moreover we write $m = |T|$, $\mathbf{s} = \mathsf{mvar}(T)$ and $\mathbf{u} = \mathbf{x} \setminus \mathbf{s}$. The ideal $\mathsf{sat}(T)$ enjoys several properties. First, its zero-set equals $\overline{W(T)}$. Second, the ideal $\mathsf{sat}(T)$ is unmixed with dimension $n - m$. Moreover, any prime ideal $\mathfrak{p}$ associated to $\mathsf{sat}(T)$ satisfies $\mathfrak{p} \cap \mathbf{k}[\mathbf{u}] = \langle 0 \rangle$.

Given $P \in \mathbf{k}[\mathbf{x}]$ the *pseudo-remainder* (resp. *iterated resultant*) of P w.r.t. T, denoted by $\mathsf{prem}(P, T)$ (resp. $\mathsf{iterRes}(P, T)$) is defined as follows. If $P \in \mathbf{k}$ or no variables of P is algebraic w.r.t. T, then $\mathsf{prem}(P, T) = P$ (resp. $\mathsf{iterRes}(P, T) = P$). Otherwise, we set $\mathsf{prem}(P, T) = \mathsf{prem}(R, T_{<v})$ (resp. $\mathsf{iterRes}(P, T) = \mathsf{iterRes}(R, T_{<v})$) where v is the largest variable of P which is algebraic w.r.t. T and R is the pseudo-remainder (resp. resultant) of P and T_v w.r.t. v. We have: P is null (resp. regular) w.r.t. $\mathsf{sat}(T)$ if and only if $\mathsf{prem}(P, T) = 0$ (resp. $\mathsf{iterRes}(P, T) \neq 0$).

Regular GCD. Let I be the ideal generated by $\sqrt{\mathsf{sat}(T)}$ in $\mathbf{k}(\mathbf{u})[\mathbf{s}]$. Then $\mathbb{L}(T) := \mathbf{k}(\mathbf{u})[\mathbf{s}]/I$ is a direct product of fields. It follows that every pair of univariate polynomials $P, Q \in \mathbb{L}(T)[y]$ possesses a GCD in the sense of [67]. The following GCD notion [65] is more convenient since it avoids considering radical ideals. Let $T \subset \mathbf{k}[x_1, \ldots, x_n]$ be a regular chain and let $P, Q \in \mathbf{k}[\mathbf{x}, y]$ be non-constant polynomials both with main variable y. Assume that the initials of P and Q are regular modulo $\mathsf{sat}(T)$. A non-zero polynomial $G \in \mathbf{k}[\mathbf{x}, y]$ is a *regular GCD* of P, Q *w.r.t.* T if these conditions hold:

(1) $\mathsf{lc}(G, y)$ is regular with respect to $\mathsf{sat}(T)$;

(2) there exist $u, v \in \mathbf{k}[\mathbf{x}, y]$ such that $g - up - vt \in \mathsf{sat}(T)$;

(3) if $\mathsf{deg}(G, y) > 0$ holds, then $\langle P, Q \rangle \subseteq \mathsf{sat}(T \cup G)$.

In this case, the polynomial G has several properties. First, it is regular with respect to $\mathsf{sat}(T)$. Moreover, if $\mathsf{sat}(T)$ is radical and $\mathsf{deg}(G, y) > 0$ holds, then the ideals $\langle P, Q\rangle$ and $\langle G\rangle$ of $\mathbb{L}(T)[y]$ are equal, so that G is a GCD of (P, Q) w.r.t. T in the sense of [67]. The notion of a regular GCD can be used to compute intersections of algebraic varieties. As an example we will use Formula (3.1) which follows from Theorem 32 in [65]. Assume that the regular chain T is simply $\{R\}$ where $R = \mathsf{res}(P, Q, y)$, for $R \notin \mathbf{k}$, and let H be the product of the initials of P and Q. Then, we have:

$$V(P, Q) = \overline{W(R, G)} \ \cup \ V(H, P, Q). \tag{3.1}$$

Splitting. Two polynomials P, Q may not necessarily admit a regular GCD w.r.t. a regular chain T, unless $\mathsf{sat}(T)$ is prime, see Example 1 in Section 3.3. However, if T "splits" into several regular chains, then P, Q may admit a regular GCD w.r.t. each of them. This requires a notation. For non-empty regular chains $T, T_1, \ldots, T_e \subset \mathbf{k}[\mathbf{x}]$ we write $T \longrightarrow (T_1, \ldots, T_e)$ whenever

$$\sqrt{\mathsf{sat}(T)} = \sqrt{\mathsf{sat}(T_1)} \cap \cdots \cap \sqrt{\mathsf{sat}(T_e)},$$

$\mathsf{mvar}(T) = \mathsf{mvar}(T_i)$ and $\mathsf{sat}(T) \subseteq \mathsf{sat}(T_i)$ hold for all $1 \leq i \leq e$. Observe that during splitting any polynomial H regular w.r.t $\mathsf{sat}(T)$ is also regular w.r.t. $\mathsf{sat}(T_i)$ for all $1 \leq i \leq e$.

3.2.2 Fundamental operations on regular chains

We list below the specifications of the fundamental operations on regular chains used in this and the following chapter. The names and specifications of these operations are the same as in the `RegularChains` library [48] in MAPLE.

Regularize. For regular chain $T \subset \mathbf{k}[\mathbf{x}]$ and polynomial $P \in \mathbf{k}[\mathbf{x}]$, the operation $\mathsf{Regularize}(P, T)$ returns regular chains $T_1, \ldots, T_e$ of $\mathbf{k}[\mathbf{x}]$ such that, for each $1 \leq i \leq e$, P is either zero or regular modulo $\mathsf{sat}(T_i)$ and we have $T \longrightarrow (T_1, \ldots, T_e)$.

RegularGcd. Let T be a regular chain and let $P, Q \in \mathbf{k}[\mathbf{x}, y]$ be non-constant with $\mathsf{mvar}(P) = \mathsf{mvar}(Q) = y \notin \mathsf{mvar}(T)$ and such that both $\mathsf{init}(P)$ and $\mathsf{init}(Q)$ are regular w.r.t. $\mathsf{sat}(T)$. Then, the operation $\mathsf{RegularGcd}(P, Q, T)$ returns a sequence $(G_1, T_1), \ldots, (G_e, T_e)$, called a *regular GCD sequence*, where $G_1, \ldots, G_e$ are polyno-

mials and $T_1, \ldots, T_e$ are regular chains of $\mathbf{k}[\mathbf{x}]$, such that $T \longrightarrow (T_1, \ldots, T_e)$ holds and G_i is a regular GCD of P, Q w.r.t. T_i for all $1 \leq i \leq e$.

NormalForm. Let T be a zero-dimensional normalized regular chain, that is, a regular chain whose saturated ideal is zero-dimensional and whose initials are all in the base field $\mathbf{k}$. Observe that T is a lexicographic Gröbner basis. Then, for $P \in \mathbf{k}[\mathbf{x}]$, the operation $\mathsf{NormalForm}(P, T)$ returns the *normal form* of P w.r.t. T in the sense of Gröbner bases.

3.2.3 Subresultants

We follow the presentation of [25], [85] and [30].

Determinantal polynomial. Let $\mathbb{B}$ be a commutative ring with identity and let $m \leq n$ be positive integers. Let M be a $m \times n$ matrix with coefficients in $\mathbb{B}$. Let M_i be the square submatrix of M consisting of the first $m-1$ columns of M and the i-th column of M, for $i = m \cdots n$; let $\det M_i$ be the determinant of M_i. We denote by dpolM the element of $\mathbb{B}[y]$, called the *determinantal polynomial* of M, given by

$$\mathsf{dpol}(M) = \det M_m y^{n-m} + \det M_{m+1} y^{n-m-1} + \cdots + \det M_n.$$

Note that if $\mathsf{dpol}(M)$ is not zero then its degree is at most $n - m$. Let $P_1, \ldots, P_m$ be polynomials of $\mathbb{B}[y]$ of degree less than n. We denote by $\mathsf{mat}(P_1, \ldots, P_m)$ the $m \times n$ matrix whose i-th row contains the coefficients of P_i, sorted in order of decreasing degree, and such that P_i is treated as a polynomial of degree $n-1$. We denote by $\mathsf{dpol}(P_1, \ldots, P_m)$ the determinantal polynomial of $\mathsf{mat}(P_1, \ldots, P_m)$.

Subresultant. Let $P, Q \in \mathbb{B}[y]$ be non-constant polynomials of respective degrees p, q with $q \leq p$. Let d be an integer with $0 \leq d < q$. Then the d-th *subresultant* of P and Q, denoted by $S_d(P, Q)$, is

$$S_d(P, Q) = \mathsf{dpol}(y^{q-d-1}P, y^{q-d-2}P, \ldots, P, y^{p-d-1}Q, \ldots, Q).$$

This is a polynomial which belongs to the ideal generated by P and Q in $\mathbb{B}[y]$. In particular, $S_0(P, Q)$ is $\mathsf{res}(P, Q)$, the resultant of P and Q. Observe that if $S_d(P, Q)$ is not zero then its degree is at most d. When $S_d(P, Q)$ has degree d, it is said *non-defective* or *regular*; when $S_d(P, Q) \neq 0$ and $\mathsf{deg}(S_d(P, Q)) < d$, $S_d(P, Q)$ is said

defective. We denote by s_d the coefficient of $S_d(P,Q)$ in y^d. For convenience, we extend the definition to the q-th subresultant as follows:

$$S_q(P,Q) = \begin{cases} \gamma(Q)Q, & \text{if } p > q \text{ or } \mathsf{lc}(Q) \in \mathbb{B} \text{ is regular} \\ \text{undefined}, & \textit{otherwise} \end{cases}$$

where $\gamma(Q) = \mathsf{lc}(Q)^{p-q-1}$. Note that when p equals q and $\mathsf{lc}(Q)$ is a regular element in $\mathbb{B}$, $S_q(P,Q) = \mathsf{lc}(Q)^{-1}Q$ is in fact a polynomial over the total fraction ring of $\mathbb{B}$. We call *specialization property of subresultants* the following statement. Let $\mathbb{D}$ be another commutative ring with identity and Ψ a ring homomorphism from $\mathbb{B}$ to $\mathbb{D}$ such that $\Psi(\mathsf{lc}(P)) \neq 0$ and $\Psi(\mathsf{lc}(Q)) \neq 0$. Then

$$S_d(\Psi(P), \Psi(Q)) = \Psi(S_d(P,Q)).$$

Divisibility relations of subresultants. Subresultants $S_{q-1}(P,Q)$, $S_{q-2}(P,Q)$, …, $S_0(P,Q)$ satisfy relations which induce an Euclidean-like algorithm for computing them. Following [25] we first assume that $\mathbb{B}$ is an integral domain. For convenience, we simply write S_d instead of $S_d(P,Q)$ for each d. We write $A \sim B$ for $A, B \in \mathbb{B}[y]$ whenever they are associated over $\mathsf{fr}(\mathbb{B})$, the field of fractions of $\mathbb{B}$. Then for $d = q-1, \ldots, 1$, we have:

(r_{q-1}) $S_{q-1} = \mathsf{prem}(P, -Q)$, the pseudo-remainder of P by $-Q$,

$(r_{<q-1})$ if $S_{q-1} \neq 0$, with $e = \mathsf{deg}(S_{q-1})$, then the following holds:

$$\mathsf{prem}(Q, -S_{q-1}) = \mathsf{lc}(Q)^{(p-q)(q-e)+1} S_{e-1},$$

(r_e) if $S_{d-1} \neq 0$, with $e = \mathsf{deg}(S_{d-1}) < d-1$, thus S_{d-1} is defective, and we have

(1) $\mathsf{deg}(S_d) = d$, thus S_d is non-defective,

(2) $S_{d-1} \sim S_e$ and $\mathsf{lc}(S_{d-1})^{d-e-1} S_{d-1} = {s_d}^{d-e-1} S_e$, thus S_e is non-defective,

(3) $S_{d-2} = S_{d-3} = \cdots = S_{e+1} = 0$,

(r_{e-1}) if both S_d and S_{d-1} are nonzero, with respective degrees d and e then we have $\mathsf{prem}(S_d, -S_{d-1}) = \mathsf{lc}(S_d)^{d-e+1} S_{e-1}$.

We consider now the case where $\mathbb{B}$ is an arbitrary commutative ring, following Theorem 4.3 in [30]. If S_d, S_{d-1} are nonzero, with respective degrees d and e and if s_d is

regular in $\mathbb{B}$ then we have

$$\mathsf{lc}(S_{d-1})^{d-e-1} S_{d-1} = {s_d}^{d-e-1} S_e.$$

Moreover, there exists $C_d \in \mathbb{B}[y]$ such that

$$(-1)^{d-1} \mathsf{lc}(S_{d-1}) s_e S_d + C_d S_{d-1} = \mathsf{lc}(S_d)^2 S_{e-1}.$$

In addition $S_{d-2} = S_{d-3} = \cdots = S_{e+1} = 0$ also holds.

From these formula we derive the following observation to which we will refer as the *block structure of subresultants*. Let S_i, S_j, S_k be three non-zero subresultants with indices $q \geq i > j > k \geq 0$. Assume that for all $\ell = i-1, \ldots, j+1, j-1, \ldots, k+1$ we have $S_\ell = 0$. Assume that S_j is defective. Then S_i is non-defective and we have $j = i - 1$. Moreover S_k is non-defective and we have $S_j \sim S_k$. Observe also that the non-zero subresultant S_d of smallest index d, sometimes called the *last subresultant* of P and Q and denoted by $\mathsf{lsr}(P, Q)$, is a non-defective subresultant.

3.3 Subresultants and regular GCDs

Throughout this section, we assume $n \geq 1$ and we consider $P, Q \in \mathbf{k}[x_1, \ldots, x_{n+1}]$ non-constant polynomials with the same main variable $y := x_{n+1}$ and such that $p := \mathsf{deg}(P, y) \geq q := \mathsf{deg}(Q, y)$ holds. We denote by R the resultant of P and Q w.r.t. y. Let $T \subset \mathbf{k}[x_1, \ldots, x_n]$ be a non-empty regular chain such that $R \in \mathsf{sat}(T)$ and the initials of P, Q are regular w.r.t. $\mathsf{sat}(T)$. We denote by $\mathbb{A}$ and $\mathbb{B}$ the rings $\mathbf{k}[x_1, \ldots, x_n]$ and $\mathbf{k}[x_1, \ldots, x_n]/\mathsf{sat}(T)$, respectively. Let Ψ be both the canonical ring homomorphism from $\mathbb{A}$ to $\mathbb{B}$ and the ring homomorphism it induces from $\mathbb{A}[y]$ to $\mathbb{B}[y]$. For $0 \leq j \leq q$, we denote by S_j the j-th subresultant of P, Q in $\mathbb{A}[y]$.

Let d be an index in the range $1 \cdots q$ such that $S_j \in \mathsf{sat}(T)$ for all $0 \leq j < d$. Lemma 13 and Lemma 14 exhibit conditions under which S_d is a regular GCD of P and Q w.r.t. T. Lemma 11 and Lemma 12 investigate the properties of S_d when $\mathsf{lc}(S_d, y)$ is regular modulo $\mathsf{sat}(T)$ and $\mathsf{lc}(S_d, y) \in \mathsf{sat}(T)$ respectively.

Lemma 11. *If $\mathsf{lc}(S_d, y)$ is regular modulo sat(T), then the polynomial S_d is a non-defective subresultant of P and Q over $\mathbb{A}$. Consequently, $\Psi(S_d)$ is a non-defective subresultant of $\Psi(P)$ and $\Psi(Q)$ in $\mathbb{B}[y]$.*

Proof. When $d = q$ holds, we are done. Assume $d < q$. Suppose S_d is defective, that is, $\mathsf{deg}(S_d, y) = e < d$. According to item (r_e) in the divisibility relations of

subresultants, there exists a non-defective subresultant S_{d+1} such that

$$\mathsf{lc}(S_d, y)^{d-e} S_d = s_{d+1}^{d-e} S_e,$$

where s_{d+1} is the leading coefficient of S_{d+1} in y. By our assumptions, S_e belongs to $\mathsf{sat}(T)$, thus $\mathsf{lc}(S_d, y)^{d-e} S_d \in \mathsf{sat}(T)$ holds. It follows from the fact $\mathsf{lc}(S_d, y)$ is regular modulo $\mathsf{sat}(T)$ that S_d is also in $\mathsf{sat}(T)$. However the fact that $\mathsf{lc}(S_d, y) = \mathsf{init}(S_d)$ is regular modulo $\mathsf{sat}(T)$ also implies that S_d is regular modulo $\mathsf{sat}(T)$. A contradiction. □

Lemma 12. *If* $\mathsf{lc}(S_d, y)$ *is contained in* $\mathsf{sat}(T)$*, then all the coefficients of* S_d *regarded as a univariate polynomial in* y *are nilpotent modulo* $\mathsf{sat}(T)$*.*

Proof. If the leading coefficient $\mathsf{lc}(S_d, y)$ is in $\mathsf{sat}(T)$, then $\mathsf{lc}(S_d, y) \in \mathfrak{p}$ holds for all the associated primes $\mathfrak{p}$ of $\mathsf{sat}(T)$. By the Block Structure Theorem of subresultants (Theorem 7.9.1 of [58]) over an integral domain $\mathbf{k}[x_1, \ldots, x_{n-1}]/\mathfrak{p}$, S_d must belong to $\mathfrak{p}$. Hence we have $S_d \in \sqrt{\mathsf{sat}(T)}$. Indeed, in a commutative ring, the radical of an ideal equals the intersection of all its associated primes. Thus S_d is nilpotent modulo $\mathsf{sat}(T)$. It follows from Exercise 2 of [5] that all the coefficients of S_d in y are also nilpotent modulo $\mathsf{sat}(T)$. □

Lemma 12 implies that, whenever $\mathsf{lc}(S_d, y) \in \mathsf{sat}(T)$ holds, the polynomial S_d will vanish on all the components of $\mathsf{sat}(T)$ after splitting T sufficiently. This is the key reason why Lemma 11 and Lemma 12 can be applied for computing regular GCDs. Indeed, up to splitting via the operation Regularize, one can always assume that either $\mathsf{lc}(S_d, y)$ is regular modulo $\mathsf{sat}(T)$ or $\mathsf{lc}(S_d, y)$ belongs to $\mathsf{sat}(T)$. Hence, up to splitting, one can assume that either $\mathsf{lc}(S_d, y)$ is regular modulo $\mathsf{sat}(T)$ or S_d belongs to $\mathsf{sat}(T)$. This leads to the following notion of a candidate regular GCD, which plays a central role for computing the regular GCD sequence of P and Q.

Definition 5 (Candidate regular GCD)**.** *Let* S_d *be the last nonzero subresultant of* P *and* Q *such that* $S_d \notin \mathsf{sat}(T)$*. Then* S_d *is called a candidate regular GCD of* P*,* Q *with respect to* T*, if the leading coefficient of* S_d *in* y *is regular modulo* $\mathsf{sat}(T)$*.*

Example 6. *If* $\mathsf{lc}(S_d, y)$ *is not regular modulo* $\mathsf{sat}(T)$ *then* S_d *may be defective. Consider for instance the polynomials* $P = x_3^2 x_2^2 - x_1^4$ *and* $Q = x_1^2 x_3^2 - x_2^4$ *in* $\mathbb{Q}[x_1, x_2, x_3]$*. We have* $\mathsf{prem}(P, -Q) = (x_1^6 - x_2^6)$ *and* $R = (x_1^6 - x_2^6)^2$*. Let* $T = \{R\}$*. The last subresultant of* P, Q *modulo* $\mathsf{sat}(T)$ *is* $\mathsf{prem}(P, -Q)$*, which has degree 0 w.r.t* x_3*, although its index is* 1*. Note that* $\mathsf{prem}(P, -Q)$ *is nilpotent modulo* $\mathsf{sat}(T)$*.*

In what follows, we give sufficient conditions for the subresultant S_d to be a regular GCD of P and Q w.r.t. T. When $\mathsf{sat}(T)$ is a radical ideal, Lemma 14 states that the assumptions of Lemma 11 are sufficient. This lemma validates the search for a regular GCD of P and Q w.r.t. T in a bottom-up style, from S_0 up to S_ℓ for some ℓ.

Lemma 13 covers the case where $\mathsf{sat}(T)$ is not radical and states that S_d is a regular GCD of P and Q modulo T, provided that S_d satisfies the conditions of Lemma 11 and provided that, for all $d < k \leq q$, the coefficient $\mathsf{coeff}(S_k, y^k)$ [1] is either null or regular modulo $\mathsf{sat}(T)$. This is our first main theoretical result.

Lemma 13. *We reuse the notations and assumptions of Lemma 11. Then S_d is a regular GCD of P and Q modulo $\mathsf{sat}(T)$, if S_d is a candidate regular GCD and for all $d < k \leq q$, the coefficient s_k of y^k in S_k is either null or regular modulo $\mathsf{sat}(T)$.*

Proof. There are three conditions to satisfy for S_d to be a regular GCD of P and Q modulo $\mathsf{sat}(T)$:

(1) $\mathsf{lc}(S_d)$ is regular modulo $\mathsf{sat}(T)$;

(2) there exists polynomials u and v such that $S_d - uP - vQ \in \mathsf{sat}(T)$; and

(3) both P and Q are in $\mathcal{I} := \mathsf{sat}(T \cup \{S_d\})$.

We write $\Psi(r)$ as $\bar{r}$ for brevity [2], and will prove the lemma in three steps.

Claim 1: If Q and S_{q-1} are in $\mathcal{I}$, then S_d is a regular GCD of P, Q modulo $\mathsf{sat}(T)$.

The properties of S_d imply Conditions (1), (2) and we only need to show that the Condition (3) also holds. If $d = q$ holds, then $S_{q-1} \in \mathsf{sat}(T)$ and we are done. Otherwise, $S_{q-1} = \mathsf{prem}(P, -Q)$ is not null modulo $\mathsf{sat}(T)$, because $\bar{S}_{q-1} = 0$ implies that all subresultants of $\bar{P}$ and $\bar{Q}$ with index less than q vanish over $\mathbb{B}$. By assumption, both Q and $S_{q-1} = \mathsf{prem}(P, -Q)$ are in $\mathcal{I}$, P is also in $\mathcal{I}$, since $\mathsf{lc}(Q)$ is regular modulo $\mathsf{sat}(T)$ and is also regular modulo $\mathcal{I}$. This completes the proof of Claim 1.

To prove that Q and S_{q-1} are in $\mathsf{sat}(T)$, we define the following set of indices

$$\mathcal{J} = \{j \mid d < j < q, \mathsf{coeff}(S_j, y^j) \notin \mathsf{sat}(T)\}.$$

[1] Let $f = a_d\, x^d + \cdots + a_0$ be a nonzero univariate polynomial in x. Then the leading coefficient and the coefficient in x^d might be different. We have $\mathsf{coeff}(f, x^i) = a_i$ for all $0 \leq i \leq d$, and the leading coefficient of f is $\mathsf{lc}(f) = a_s$ where s is the largest index such $a_s \neq 0$. Hence $\mathsf{coeff}(f, x^d)$ may differ from $\mathsf{lc}(f)$.

[2] The degree of $\bar{S}_k$ may be less than the degree of S_k, since its leading coefficient could be in $\mathsf{sat}(T)$. Hence, $\overline{\mathsf{lc}(S_k)}$ may differ from $\mathsf{lc}(\bar{S}_k)$. We carefully distinguish them when the leading coefficient of a subresultant is not regular in $\mathbb{B}$.

By assumption, $\mathsf{coeff}(S_j, y^j)$ is regular modulo $\mathsf{sat}(T)$ for each $j \in \mathcal{J}$. Our arguments rely on the Block Structure Theorem (BST) over an arbitrary ring [30] and Ducos' subresultant algorithm [25, 65] along with the specialization property of subresultants.

Claim 2: If $\mathcal{J} = \emptyset$, then $S_i \in \mathcal{I}$ holds for all $d < i \leq q$.

Indeed, the BST over $\mathbb{B}$ implies that there exists *at most* one subresultant S_j such that $d < j < q$ and $S_j \notin \mathsf{sat}(T)$. Therefore all but S_{q-1} are in $\mathsf{sat}(T)$, and thus $\bar{S}_{q-1}$ is defective of degree d. More precisely, the BST over $\mathbb{B}$ implies

$$\mathsf{lc}(\bar{S}_{q-1})^e S_{q-1} \equiv \mathsf{lc}(S_q)^e S_d \mod \mathsf{sat}(T) \tag{3.2}$$

for some integer $e \geq 0$. According to Relation (3.2), $\mathsf{lc}(\bar{S}_{q-1})$ is regular in $\mathbb{B}$. Hence, we have $S_{q-1} \in \mathcal{I}$. By the definition of S_d, we have $\mathsf{prem}(\bar{S}_q, -\bar{S}_{q-1}, y) \in \mathsf{sat}(T)$ which implies $S_q \in \mathcal{I}$. This completes the proof of Claim 2.

Now we consider the case $\mathcal{J} \neq \emptyset$. Write $\mathcal{J}$ explicitly as $\mathcal{J} = \{j_0, j_1, \ldots, j_{\ell-1}\}$, with $\ell = |\mathcal{J}|$ and we assume $j_0 < j_1 < \cdots < j_{\ell-1}$. For convenience, we write $j_\ell := q$. For each integer k satisfying $0 \leq k \leq \ell$ we denote by $\mathcal{P}_k$ the following property:

$$S_i \in \mathcal{I}, \quad \text{for all} \quad d < i \leq j_k.$$

Claim 3: The property $\mathcal{P}_k$ holds for all $0 \leq k \leq \ell$.

We proceed by induction on $0 \leq k \leq \ell$. The base case is $k = 0$. We need to show $S_i \in \mathcal{I}$ for all $d < i \leq j_0$. By the definition of j_0, $\bar{S}_{j_0}$ is a non-defective subresultant of $\bar{P}$ and $\bar{Q}$, and $\mathsf{coeff}(S_i, y^i)$ is in $\mathsf{sat}(T)$ for all $d < i < j_0$. By the BST over $\mathbb{B}$, there is *at most* one $d < i < j_0$ such that $S_i \notin \mathsf{sat}(T)$. If no such a subresultant exists, then we know that $\mathsf{prem}(\bar{S}_{j_0}, -\bar{S}_d)$ is in $\mathsf{sat}(T)$. Consequently, $S_{j_0} \in \mathcal{I}$ holds, which implies $S_j \in \mathcal{I}$ for all $d < i \leq j_0$. On the other hand, if S_{i_0} is not in $\mathsf{sat}(T)$ for some $d < i_0 < j_0$, then $\bar{S}_{i_0}$ is similar to $\bar{S}_d$ over $\mathbb{B}$. To be more precise, we have

$$\mathsf{lc}(\bar{S}_{i_0})^e \bar{S}_{i_0} \equiv \mathsf{lc}(\bar{S}_{j_0})^e \bar{S}_d \mod \mathsf{sat}(T) \tag{3.3}$$

for some integer $e \geq 0$. With the same reasoning as in the case $\mathcal{J} = \emptyset$, we know that $\mathsf{lc}(\bar{S}_{i_0})$ is regular modulo $\mathsf{sat}(T)$ and we deduce that $S_{i_0} \in \mathcal{I}$ holds. Also, we have $\mathsf{prem}(\bar{S}_{j_0}, -\bar{S}_{i_0}) \in \mathsf{sat}(T)$, by definition of S_d. This implies $S_{j_0} \in \mathcal{I}$ from the fact that $\mathsf{lc}(\bar{S}_{i_0})$ is regular modulo $\mathsf{sat}(T)$ (and thus regular modulo $\mathcal{I}$). Hence, we have $S_i \in \mathcal{I}$ for all $d < i \leq j_0$, as desired. Therefore the property $\mathcal{P}_k$ holds for $k = 0$.

Now we assume that the property $\mathcal{P}_{k-1}$ holds for some $1 \leq k \leq \ell$. We prove that

$\mathcal{P}_k$ also holds. According to the BST over $\mathbb{B}$, there exists *at most* one subresultant between $\bar{S}_{j_{k-1}}$ and $\bar{S}_{j_k}$, both of which are non-defective subresultants of $\bar{P}$ and $\bar{Q}$. If $S_i \in \mathsf{sat}(T)$ holds for all $j_{k-1} < i < j_k$, then we have

$$\mathsf{prem}(\bar{S}_{j_k}, -\bar{S}_{j_{k-1}}) \equiv \mathsf{lc}(\bar{S}_{j_k})^e \bar{S}_u \mod \mathsf{sat}(T)$$

for some $d \leq u < j_{k-1}$ and some integer $e \geq 0$. Thus, $\mathsf{prem}(\bar{S}_{j_k}, -\bar{S}_{j_{k-1}}) \in \mathcal{I}$ by our induction hypothesis, and consequently, $S_{j_k} \in \mathcal{I}$ holds. On the other hand, if all subresultants S_i (for $j_{k-1} < i < j_k$) but S_{i_k} (for some index i_k such that $j_{k-1} < i_k < j_k$) are in $\mathsf{sat}(T)$, then $\bar{S}_{i_k}$ is similar to $\bar{S}_{j_{k-1}}$ over $\mathbb{B}$, that is,

$$\mathsf{lc}(\bar{S}_{i_k})^e \bar{S}_{i_k} \equiv \mathsf{lc}(\bar{S}_{j_k})^e \bar{S}_{j_{k-1}} \mod \mathsf{sat}(T) \tag{3.4}$$

for some integer $e \geq 0$. By Relation (3.4), $\mathsf{lc}(\bar{S}_{i_k})$ is regular modulo $\mathsf{sat}(T)$, and thus is regular modulo $\mathcal{I}$. Using Relation (3.4) again, we have $S_{i_k} \in \mathcal{I}$, since $S_{j_{k-1}}$ is in $\mathcal{I}$. Meanwhile, we have

$$\mathsf{prem}(\bar{S}_{j_k}, -\bar{S}_{i_k}) \equiv \mathsf{lc}(\bar{S}_{j_k})^e \bar{S}_u \mod \mathsf{sat}(T)$$

for some $d \leq u < j_{k-1}$ and some integer $e \geq 0$. By the induction hypothesis, we deduce $S_u \in \mathcal{I}$, which implies $S_{j_k} \in \mathcal{I}$ together with the fact that $\mathsf{lc}(\bar{S}_{i_k})$ is regular modulo $\mathcal{I}$. This shows that $S_i \in \mathcal{I}$ holds for all $d < i \leq j_k$. Therefore, property $\mathcal{P}_k$ holds.

Finally, we apply Claim 3 with $k = \ell$, leading to $S_i \in \mathcal{I}$ for all $d < i \leq j_\ell = q$, which completes the proof of our lemma. □

The consequence of the above corollary is that we ensure that S_d is a regular GCD after checking that the leading coefficients of all non-defective subresultants above S_d, are either null or regular modulo $\mathsf{sat}(T)$. Therefore, one may be able to conclude that S_d is a regular GCD simply after checking the coefficients "along the diagonal" of the pictorial representation of the subresultants of P and Q.

On the left of Figure 3.1, P and Q have five nonzero subresultants over $\mathbf{k}[\mathbf{x}]$, four of which are non-defective and one of which is defective.Let T be a regular chain in $\mathbf{k}[\mathbf{x}]$ such that $\mathsf{lc}(P)$ and $\mathsf{lc}(Q)$ are regular modulo $\mathsf{sat}(T)$. Further, we assume that $\mathsf{lc}(S_1)$ and $\mathsf{lc}(S_4)$ are regular modulo $\mathsf{sat}(T)$, however, $\mathsf{lc}(S_6)$ is in $\mathsf{sat}(T)$. The right hand side is a possible configuration of the subresultant chain of $\bar{P}$ and $\bar{Q}$. In the proof of Claim 3, the set $\mathcal{J}$ is $\{j_0 = 4\}$ and $j_1 = 7$, whereas $i_0 = 2$ and $i_1 = 6$ are the

indices of defective subresultants over $\mathbf{k}[\mathbf{x}]/\mathsf{sat}(T)$. In this case, S_1 is a regular GCD of P and Q modulo $\mathsf{sat}(T)$.

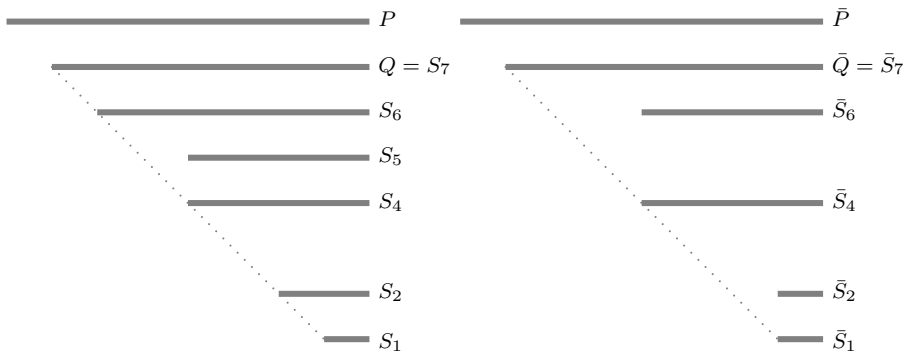

Figure 3.1: A possible configuration of the subresultant chain of P and Q.

Lemma 14. *With the assumptions of Lemma 11, assume* $\mathsf{sat}(T)$ *radical. Then,* S_d *is a regular GCD of* P, Q *w.r.t.* T*.*

Proof. As for Lemma 13, it suffices to check that P, Q belong to $\mathsf{sat}(T \cup \{S_d\})$. Let $\mathfrak{p}$ be any prime ideal associated with $\mathsf{sat}(T)$. Define $\mathbb{D} = \mathbf{k}[x_1, \ldots, y]/\mathfrak{p}$ and let $\mathbb{L}$ be the fraction field of the integral domain $\mathbb{D}$. Clearly S_d is the last subresultant of $P,\ Q$ in $\mathbb{D}[y]$ and thus in $\mathbb{L}[y]$. Hence S_d is a GCD of $P,\ Q$ in $\mathbb{L}[y]$. Thus S_d divides $P,\ Q$ in $\mathbb{L}[y]$ and pseudo-divides $P,\ Q$ in $\mathbb{D}[y]$. Therefore $\mathsf{prem}(P, S_d)$ and $\mathsf{prem}(Q, S_d)$ belong to $\mathfrak{p}$. Finally $\mathsf{prem}(P, S_d)$ and $\mathsf{prem}(Q, S_d)$ belong to $\mathsf{sat}(T)$. Indeed, $\mathsf{sat}(T)$ being radical, it is the intersection of its associated primes. □

3.4 A regular GCD algorithm

Following the notations and assumptions of Section 3.3, we propose an algorithm to compute a regular GCD sequence of P, Q w.r.t. T. as specified in Section 3.2.2. Then, we explain how to relax the assumption $R \in \mathsf{sat}(T)$. First, the subresultants of P, Q in $\mathbb{A}[y]$ are assumed to be known. We explain in Section 4.2 how we compute them in our implementation. Secondly, we rely on the **Regularize** operation specified in Section 3.2.2. Lastly, we inspect the subresultant chain of P, Q in $\mathbb{A}[y]$ in a bottom-up manner. Therefore, we view $S_1, S_2, \ldots$ as successive candidates and apply either Lemma 14, (if $\mathsf{sat}(T)$ is known to be radical) or Lemma 13.

3.4.1 Case where $R \in \mathsf{sat}(T)$.

By virtue of Lemma 11 and Lemma 12 there exists regular chains $T_1, \ldots, T_e \subset \mathbf{k}[\mathbf{x}]$ such that $T \longrightarrow (T_1, \ldots, T_e)$ holds and for each $1 \leq i \leq e$ there exists an index $1 \leq d_i \leq q$ such that the leading coefficient $\mathsf{lc}(S_{d_i}, y)$ of the subresultant S_{d_i} is regular modulo $\mathsf{sat}(T_i)$ and $S_j \in \mathsf{sat}(T_i)$ for all $0 \leq j < d_i$. Such regular chains can be computed using the operation Regularize. If each $\mathsf{sat}(T_i)$ is radical then it follows from Lemma 14 that $(S_{d_1}, T_1), \ldots, (S_{d_e}, T_e)$ is a regular GCD sequence of P, Q w.r.t. T. In practice, when $\mathsf{sat}(\mathrm{T})$ is radical then so are all $\mathsf{sat}(T_i)$, see [9]. If some $\mathsf{sat}(T_i)$ is not known to be radical, then one can compute regular chains $T_{i,1}, \ldots, T_{i,e_i} \subset \mathbf{k}[\mathbf{x}]$ such that $T_i \longrightarrow (T_{i,1}, \ldots, T_{i,e_i})$ holds and for each $1 \leq \ell_i \leq e_i$ there exists an index $1 \leq d_{\ell_i} \leq q$ such that Lemma 13 applies and shows that the subresultant $S_{d_{\ell_i}}$ is regular GCD of P, Q w.r.t. T_{i,ℓ_i}. Such computation relies again on Regularize. The complete procedure of computing regular GCD sequence is given by the algorithm RGSZR.

Theorem 5. *The algorithm* RGSZR *terminates and computes a regular GCD sequence of P and Q with respect to T.*

Proof. We need to show the termination of two while-loops in the algorithm. The first one is the while-loop from Line **4** to Line **15**. Observe that if an item $[i, C]$ out of $Tasks$ satisfies $i = \mathsf{mdeg}(Q)$, then both S_i and c_i are regular modulo $\mathsf{sat}(C)$. Then no true splitting will happen (i.e. $C = D$ at Line **11**) while calling $\mathsf{Regularize}(S_i, C)$. Moreover, item $[i, D]$ will only be inserted into $Candidates$ (Line **15**). In other words, no item $[i, D]$ with $i > \mathsf{mdeg}(Q)$ will appear during the computation. Since only items $[i + 1, D]$, replacing the item $[i, C]$ from $Tasks$, can be inserted back at Line **9** and Line **13**, this while-loop terminates eventually. The second while-loop is from Line **22** to Line **29**. According to Line **24** and **25**, no item $[j, D]$ with $j > \mathsf{mdeg}(Q)$ will appear during the computation. Hence the latter while-loop terminates as well. This completes the proof of the termination of the algorithm.

Now we prove the correctness of the algorithm. During each iteration of the while-loop from Line **4** to Line **15**, from the specification of Regularize, regular chains in $\mathsf{Regularize}(S_i, C)$ form a splitting of C. This implies a loop invariant: regular chains in $Tasks$ and $Candidates$ form a splitting of T. What we need to show is: each item $[i, D]$ of $Candidates$ satisfies that S_i is a candidate regular GCD of P, Q w.r.t D and $\mathsf{lc}(S_i, y)$ is regular modulo $\mathsf{sat}(D)$.

We first prove a loop invariant for items in $Tasks$: *During each iteration of the first while-loop, each item $[i, C]$ in $Tasks$ satisfies $S_k \in \mathsf{sat}(C)$ for all $k < i$.*

Algorithm 3: RGSZR(P, Q, T)

RGSZR(P, Q, T)

Input : P and Q are polynomials $\in \mathbf{k}[\mathbf{x}][y]$ such that $\mathsf{lc}(P, y)$, $\mathsf{lc}(Q, y)$ are regular modulo $\mathsf{sat}(T)$, $\mathsf{res}P, Q, y \in \mathsf{sat}(T)$, and $\mathsf{deg}(P, y) \geq \mathsf{deg}(Q, y) > 0$

Output : a regular GCD sequence of P, Q w.r.t T

```
1  Compute subresultants S_i of P and Q in y for 1 ≤ i ≤ mdeg(Q)
   // Compute regular GCD candidates
2  Find the smallest index i such that S_i ∉ sat(T)
3  Candidates ← ∅, Tasks ← {[i, T]}
4  while Tasks ≠ ∅ do
5      Take and remove an item [i, C] out of Tasks
6      c_i ← lc(S_i, y)
7      if c_i ∈ sat(C) then
8          for D ∈ Regularize(S_i, C) do
9              Tasks ← Tasks ∪ {[i + 1, D]}
10     else
11         for D ∈ Regularize(S_i, C) do
12             if c_i ∈ sat(D) then
13                 Tasks ← Tasks ∪ {[i + 1, D]}
14             else
15                 Candidates ← Candidates ∪ {[i, D]}
   // Check all regular GCD candidates
16 if sat(T) is known to be radical then
17     for [i, C] ∈ Candidates do
18         Results ← Results ∪ {[S_i, C]}
19 else
20     for [i, C] ∈ Candidates do
21         Tasks ← {[i, C]}, Split ← ∅
22         while Tasks ≠ ∅ do
23             Take and remove an item [j, D] out of Tasks
24             if j = mdeg(Q) then
25                 Split ← Split ∪ {D}
26             else
27                 Find the smallest k > j, s.t. s_k = coeff(S_k, y^k) ∉ sat(D)
28                 for E ∈ Regularize(s_k, D) do
29                     Tasks := Tasks ∪ {[j + 1, E]}
30         for E ∈ Split do
31             Results ← Results ∪ {[S_i, E]}
32 return Results
```

Firstly, for each item $[i+1, D]$ inserted back into $Tasks$ at Line **9**, we need to show $S_i \in \mathsf{sat}(D)$ for each $D \in \mathsf{Regularize}(S_i, C)$. Since $c_i \in \mathsf{sat}(C)$, by Lemma 12, S_i is a nilpotent modulo $\mathsf{sat}(C)$, and thus S_i cannot be regular modulo $\mathsf{sat}(D)$. By the specification of $\mathsf{Regularize}$, $S_i \in \mathsf{sat}(D)$ for each D. Secondly, for each item $[i+1, D]$ inserted back into $Tasks$ at Line **13**, we know $c_i \notin \mathsf{sat}(C)$ but $c_i \in \mathsf{sat}(D)$, and need to show $S_i \in \mathsf{sat}(D)$. Lemma 12 still applies, which implies that S_i is a nilpotent modulo $\mathsf{sat}(D)$. Since D is a regular chain in $\mathsf{Regularize}(S_i, D)$, S_i must be null modulo $\mathsf{sat}(D)$. This proves the loop invariant.

Now for each item $[i, D]$ inserted into $Candidates$ at Line **15**, $c_i = \mathsf{lc}(S_i, y)$ is regular modulo $\mathsf{sat}(D)$, since we have $c_i \notin \mathsf{sat}(D)$ and $D \in \mathsf{Regularize}(S_i, C)$. It follows from the fact that $[i, C]$ is taken from $Tasks$, $S_k \in \mathsf{sat}(D)$ holds for all $k < i$. Therefore, for each $[i, D]$ in $Candidates$, S_i is a candidate regular GCD of P, Q w.r.t D, as desired.

If $\mathsf{sat}(T)$ is known to be radical, then we are done according to Lemma 14.

To finalize the proof, we show the correctness of the procedure checking each candidate regular GCD (Lines **16** to **31**). With a similar reasoning as above, during each iteration of the for-loop from Line **20** to Line **31**, all regular chains appearing in $Candidates$ and $Results$ still form a splitting of T. We only need to show that each item $[S_i, E]$ inserted into $Results$ satisfies that S_i is a regular GCD of P, Q w.r.t E.

Indeed, a key invariant of the while-loop from Line **22** to Line **29** is: each item $[j, D]$ from $Tasks$ satisfies that $\mathsf{coeff}(S_k, y^k)$ is null or regular modulo $\mathsf{sat}(D)$ for each $i < k \leq j$. This invariant is clearly maintained by Lines **27**, **28** and **29**. After finishing this while-loop, the set $Split$ consists of regular chains E such that $\mathsf{coeff}(S_k, y^k)$ is null or regular modulo $\mathsf{sat}(E)$ for each $i < k \leq \mathsf{mdeg}(Q)$. According to Lemma 13, S_i is a regular GCD of P, Q w.r.t E, which will be added into $Results$ at Line **31**. □

Recall the definition of a candidate regular GCD. Given a regular chain T in $\mathbf{k}[\mathbf{x}]$, and polynomials P, Q in $\mathbf{k}[\mathbf{x}][x_{n+1}]$ such that $\mathsf{mvar}(P) = \mathsf{mvar}(Q) = x_{n+1}$, $\mathsf{mdeg}(P) \geq \mathsf{mdeg}(Q)$, and initials of P, Q are regular modulo $\mathsf{sat}(T)$. Assume that there exists an index d in the range $1 \cdots q = \mathsf{mdeg}(Q)$ such that $S_d \notin \mathsf{sat}(T)$ and $S_j \in \mathsf{sat}(T)$ for all $0 \leq j < d$. The subresultant S_d is called a *candidate regular GCD* of P and Q w.r.t T. The following corollary is a direct consequence of Theorem 5.

Corollary 3 (Existence of regular GCDs)**.** *The subresultant S_d may not be a regular GCD of P, Q w.r.t T. However, there exists a splitting $T \to (T_1, \ldots, T_m)$ and a sequence $d_1, \ldots, d_m$ such that for each i in $1 \cdots m$, $d \leq d_i \leq q$ and S_{d_i} is a regular GCD of P, Q w.r.t. T_i.*

According to the definition of a regular GCD, S_{d_i} is a polynomial of positive degree d_i in x_{n+1} and its leading coefficient $s_{d_i} = \mathsf{lc}(S_{d_i}, x_{n+1})$ is regular modulo $\mathsf{sat}(T_i)$, which implies that S_{d_i} is of positive degree d_i modulo $\mathsf{sat}(T_i)$. In other words, regular GCD of positive degree exists in each branch of T.

3.4.2 Case where $R \notin \mathsf{sat}(T)$.

We explain how to relax the assumption $R \in \mathsf{sat}(T)$ and obtain a general algorithm for the operation RegularGcd. The principle is straightforward. Let $R = \mathsf{res}(P, Q, y)$. We call $\mathsf{Regularize}(R, T)$ obtaining regular chains $T_1, \ldots, T_e$ such that $T \longrightarrow (T_1, \ldots, T_e)$. For each $1 \leq i \leq e$, we compute a regular GCD sequence of P and Q w.r.t. T_i as follows:

(1) If $R \in \mathsf{sat}(T_i)$ holds then we proceed with Algorithm RGSZR;

(2) otherwise $R \notin \mathsf{sat}(T_i)$ holds and the resultant R is actually a regular GCD of P and Q w.r.t. T_i by definition.

Observe that when $R \in \mathsf{sat}(T_i)$ holds the subresultant chain of P and Q in y is used to compute their regular GCD w.r.t. T_i. This is one of the motivations for the implementation techniques described in Sections 4.2 and 4.3 of Chapter 4.

3.5 An example

In this section, we illustrate the algorithm RGSZR by running a nontrivial example. Let P, Q be bivariate polynomials in $\mathbb{Z}_2[a, x]$

$$P = x^6 + ax^5 + a^2x^4 + x^3 + \left(1 + a^2 + a\right)x + a^2 + a$$
$$Q = x^5 + x^4 + a^2x^3 + x^2$$

The subresultant chain of P and Q w.r.t x is

$$\left|\begin{array}{l} S_4 = s_{44}x^4 + s_{43}x^3 + s_{42}x^2 + s_{41}x + s_{40} \\ S_3 = s_{33}x^3 + s_{32}x^2 + s_{31}x + s_{30} \\ S_2 = s_{22}x^2 + s_{21}x + s_{20} \\ S_1 = s_{11}x + s_{10} \\ S_0 = s_{00} \end{array}\right.$$

where the coefficients are

$$\left|\begin{array}{l} s_{40} = (a+1)\,a \\ s_{41} = 1 + a^2 + a \\ s_{42} = a + 1 \\ s_{43} = (a+1)\,a^2 \\ s_{44} = a + 1 \end{array}\right.$$

$$\left|\begin{array}{l} s_{30} = (a+1)^4\,a \\ s_{31} = (a+1)^2\,(a^3 + a + 1) \\ s_{32} = (a+1)\,(a^3 + a + 1) \\ s_{33} = (a+1)^6 \end{array}\right.$$

$$\left|\begin{array}{l} s_{20} = (a+1)^4(1 + a^2 + a)a^4 \\ s_{21} = (a+1)^2(1 + a^2 + a)a^3(a^3 + a + 1) \\ s_{22} = (a+1)(1 + a^2 + a)(a^3 + a^2 + 1) \end{array}\right.$$

$$\left|\begin{array}{l} s_{10} = (a+1)(a^4 + a^3 + 1)(a^4 + a + 1)(1 + a^2 + a)^2 a \\ s_{11} = (a^{10} + a^8 + a^6 + a + 1)(1 + a^2 + a)^2 \end{array}\right.$$

$$\left|\begin{array}{l} s_{00} = (a+1)^2(1 + a^2 + a)^4 a^2 (a^6 + a^5 + 1) \end{array}\right.$$

Consider regular chain T consisting of a single polynomial $T = \{(a+1)^2(1+a^2+a)^4a^2\}$.

Search for candidates. Initially, the set $\mathcal{C}$ of candidates is empty.

(1) Process s_{00}.
Since $s_{00} \in \mathsf{sat}(T) = \langle (a+1)^2(1+a^2+a)^4a^2 \rangle$, proceed to the next level.

(2) Process s_{11}.
Since s_{11} is a zero-divisor modulo $\mathsf{sat}(T)$, we regularize s_{11} w.r.t T, which decomposes T into three components $T_1 = \{(a+1)^2a^2\}$, $T_2 = \{(1+a^2+a)^2\}$ and $T_3 = \{(1+a^2+a)^2\}$. Over T_1, s_{11} is regular and we add the pair $[S_1, T_1]$ into $\mathcal{C}$. Over T_2 or T_3, s_{11} is zero and we continue to the next coefficient s_{10} in S_1.

(3) Process s_{10}.
Since s_{10} is in $\mathsf{sat}(T_2)$ or $\mathsf{sat}(T_3)$, proceed to the next level.

(4) Process s_{22}.
Since s_{22} is a zero-divisor modulo $\mathsf{sat}(T_2) = \langle(1 + a^2 + a)^2\rangle$, we regularize s_{22} w.r.t T_2, which decompose T_2 into two components $T_4 = T_5 = \{1 + a^2 + a\}$. For the same reason, we decompose T_3 into two components $T_6 = T_7 = \{1 + a^2 + a\}$. For all components, we continue to the next coefficient s_{21} in S_1.

(5) Process s_{21}.
Since s_{21} is in $\mathsf{sat}(T_4)$, $\mathsf{sat}(T_5)$, $\mathsf{sat}(T_6)$ or $\mathsf{sat}(T_7)$, we continue to the next coefficient s_{20} in S_1.

(6) Process s_{20}.
Since s_{20} is in $\mathsf{sat}(T_4)$, $\mathsf{sat}(T_5)$, $\mathsf{sat}(T_6)$ or $\mathsf{sat}(T_7)$, we proceed to the next level.

(7) Process s_{33}.
Since s_{33} is regular modulo $\mathsf{sat}(T_4)$, $\mathsf{sat}(T_5)$, $\mathsf{sat}(T_6)$ or $\mathsf{sat}(T_7)$, we add pairs $[S_3, T_4]$, $[S_3, T_5]$, $[S_3, T_6]$ and $[S_3, T_7]$ into $\mathcal{C}$.

Once completed the searching for candidate regular GCDs, the set of candidate is

$$\mathcal{C} = \{[S_1, T_1], [S_3, T_4], [S_3, T_5], [S_3, T_6], [S_3, T_7]\},$$

where $T_1 = \{(a + 1)^2 a^2\}$ and $T_4 = T_5 = T_6 = T_7 = \{1 + a^2 + a\}$.

Check candidates. Initially, the set $\mathcal{G}$ of output is empty.

- Check the candidate $[S_1, T_1]$.

  (1) Process s_{22}.
  Since s_{22} is a zero-divisor modulo $\mathsf{sat}(T_1) = \langle(a + 1)^2 a^2\rangle$, we regularize s_{22} w.r.t T_1 which decomposes T_1 into three components $T_8 = \{a^2\}$ and $T_9 = T_{10} = \{a + 1\}$.

  (2) Process s_{33}.
  Since s_{33} is regular modulo $\mathsf{sat}(T_8)$, we proceed to the next level. Since s_{33} is in $\mathsf{sat}(T_9)$ or $\mathsf{sat}(T_{10})$, we proceed to the next level.

  (3) Process s_{44}.
  Since s_{44} is regular modulo $\mathsf{sat}(T_8)$, we add $[S_1, T_8]$ to $\mathcal{G}$. Since s_{44} is in $\mathsf{sat}(T_9)$ or $\mathsf{sat}(T_{10})$, we add $[S_1, T_9]$ and $[S_1, T_{10}]$ into $\mathcal{G}$.

- Check the candidate $[S_3, T_i]$ for $i = 4$, 5, 6, and 7.

(1) Process s_{44}.
Since $s_{44} = a + 1$ is regular modulo $\mathsf{sat}(T_i) = \langle 1 + a^2 + a \rangle$ for $i = 4, 5, 6,$ and 7, we add $[S_1, T_4]$, $[S_1, T_5]$, $[S_1, T_6]$, $[S_1, T_7]$ into $\mathcal{C}$.

The output. The regular GCD sequence of P and Q modulo $\mathsf{sat}(T)$ is

$$[S_1, \{a^2\}], [S_1, \{a + 1\}], [S_1, \{a + 1\}],$$
$$[S_3, \{1 + a^2 + a\}], [S_3, \{1 + a^2 + a\}], [S_3, \{1 + a^2 + a\}], [S_3, \{1 + a^2 + a\}],$$

in which there are three different regular GCDs for P and Q.

The algorithm RGSZR proceeds in a bottom-up manner. Once constructed the candidate regular GCDs, the checking phase only uses coefficients of subresultants along the diagonal. For practical problems, the regular GCDs often have degree one, which implies only $O(\mathsf{deg}(Q))$ coefficients are needed, while the total number of coefficients in the subresultant chain is $\frac{(\mathsf{deg}(Q)+1)\mathsf{deg}(Q)}{2}$. In the example, coefficients s_{40}, s_{41} and s_{42} are not used at all.

3.6 Summary

The concept of a regular GCD extends the usual notion of polynomial GCD from polynomial rings over fields to polynomial rings modulo saturated ideals of regular chains. Regular GCDs play a central role in triangular decomposition methods. Traditionally, regular GCDs are computed in a top-down manner, by adapting standard PRS techniques (Euclidean Algorithm, subresultant algorithms, ...).

In this chapter, we study carefully the relations between subresultants and regular GCD sequence of two polynomials modulo a regular chain. Our main result in this chapter is Algorithm RGSZR, in which the computation of regular GCDs is separated from the computation of subresultants. This has three benefits. First, this algorithm is well-suited to employ modular methods and fast polynomial arithmetic. Secondly, we avoid the repetition of (potentially expensive) intermediate computations. Lastly, we avoid, as much as possible, computing modulo regular chains and use polynomial computations over the base field instead, while controlling expression swell.

In the following chapters, we discuss issues surrounding the algorithm, which includes the design of subresultant chain data structure, the complexity of computing regular GCDs, efficient implementation of regular GCDs, and GPU acceleration of subresultant chain construction.

Chapter 4

Implementation and Complexity Analysis

4.1 Introduction

This chapter is a continuation of Chapter 3, with focuses on the implementation and complexity analysis. The bottom-up algorithm RGSZR provides a general framework for computing regular GCD sequence modulo a regular chain. It is well-suited to employ modular methods and fast polynomial arithmetic.

In Section 4.2 we describe our implementation for subresultant chain computation. We observe that, during the computation of triangular decomposition, whenever a regular GCD of P and Q w.r.t. T is needed, the resultant of P and Q with respect to y is likely to be computed too. This suggests to organize calculations in a way that the subresultant chain of P and Q is computed only once. To this end, we evaluate the subresultant chain of P and Q at sufficiently many values of $(x_1, \ldots, x_n)$ such that any coefficient of any subresultant P and Q can be interpolated whenever needed. In our implementation, this evaluation-interpolation scheme is based on FFT techniques. It is available in MAPLE in the module `FastArithmeticTools` of the `RegularChains` library.

The use of fast arithmetic for computing regular GCDs was proposed in [22] for regular chains with zero-dimensional radical saturated ideals. Algorithm 3 in Section 3.4 of Chapter 3, however, does not suffer from any such restrictions: the saturated ideal of T may be non-radical or of positive dimension. Algorithm 3 relies on a procedure for testing whether a polynomial is regular w.r.t the saturated ideal of

a regular chain. In Section 4.3, we propose a new algorithm for this task in dimension zero, see Algorithm 4.

Under genericity assumptions, we establish running time estimates for both Algorithms 3 and 4, see Theorem 6 and Corollary 5. We explain in Section 4.3 why these results suggest that Algorithms 3 and 4 are probably more suitable for implementation than the algorithms of [22].

The experimental results of Section 4.5 illustrate the efficiency of our algorithms. We obtain speedup factors of several orders of magnitude w.r.t. the algorithms of [65] for regular GCD computations and regularity test. Our code compares and often outperforms packages with similar specifications in MAPLE and MAGMA.

This chapter is a joint work with Xin Li and Marc Moreno Maza, based on our ISSAC 2009 article [49].

4.2 Subresultant chain computation

In this section, we report implementation techniques and complexity analysis on constructing subresultant chains. Our encoding of the subresultant chain of P, Q in $\mathbf{k}[x_1, \ldots, x_n][y]$ will be used in both our implementation and complexity results. For simplicity our analysis is restricted to the case where $\mathbf{k}$ is a finite field whose characteristic is large enough. The case where $\mathbf{k}$ is the field $\mathbb{Q}$ of rational numbers could be handled in a similar fashion, with the necessary adjustments. We follow the notations introduced in Section 3.3. However we do not assume that $R = \mathsf{res}(P, Q, y)$ necessarily belongs to the saturated ideal of the regular chain T.

One motivation for the design of the techniques presented in this chapter is the solving of systems of two equations, say $P = Q = 0$. Indeed, this can be seen as a fundamental operation in incremental methods for solving systems of polynomial equations, such as the one of [65]. We make two simple observations. Formula 3.1 p. 39 shows that solving this system reduces essentially to computing R and a regular GCD sequence of P, Q modulo $\{R\}$, when R is not constant. This is particularly true when $n = 1$ since in this case the variety $V(H, P, Q)$ is likely to be empty for generic polynomials P, Q.

The second observation is that, under the same genericity assumptions, a regular GCD G of P, Q w.r.t. $\{R\}$ is likely to exist and have degree one w.r.t. y. Therefore, once the subresultant chain of P, Q w.r.t. y is calculated, one can obtain G essentially at no additional cost. At the end of this section, we shall return to these observations and deduce complexity results from them.

The subresultant chain of P and Q is represented by homomorphic images: following [18], we evaluate $(x_1, \ldots, x_n)$ at sufficiently may points such that the subresultants of P and Q (regarded as univariate polynomials in $y = x_{n+1}$) can be computed by interpolation. To be more precise, we need some notations. Let d_i be the maximum of the degrees of P and Q in x_i, for all $i = 1, \ldots, n+1$. Observe that $b_i := 2d_i d_{n+1}$ is an upper bound for the degree of R (or any subresultant of P and Q) in x_i, for all i. Let B be the product $(b_1 + 1) \cdots (b_n + 1)$.

Specialization grid (SCube). We proceed by evaluation/interpolation; our sample points are chosen on an n-dimensional rectangular grid. We call *specialization grid* or simply *SCube* the data consisting of this grid and the values that the subresultant chain of P, Q takes at each point of this grid. This is precisely how the subresultants of P, Q are encoded in our implementation. Of course, the validity of this approach requires that our evaluation points cancel no initials of P and Q. Even though finding such points deterministically is a difficult problem, this creates no issue in practice. Whenever possible (typically, over suitable finite fields), we choose roots of unity as sample points, so that we can use FFT (or van der Hoeven's Truncated Fourier Transform [38]); otherwise, standard fast evaluation/interpolation algorithms are used, like the subproduct-tree technique [34].

In order to reconstruct all subresultants of P and Q, from their SCube, one needs to perform $O(d_{n+1})$ evaluations and $O(d_{n+1}^2)$ interpolations. Since our sample points lie on a grid, the total cost (including the computation of the images of the subresultants on the grid) becomes

$$O\left(Bd_{n+1}^2 \sum_{i=1}^{n} \log(b_i)\right) \quad \text{or} \quad O\left(Bd_{n+1}^2 \sum_{i=1}^{n} \frac{\mathsf{M}(b_i)\log(b_i)}{b_i}\right),$$

depending on the choice of the sample points (see e.g. [71] for similar estimates). Here, as usual, $\mathsf{M}(b)$ stands for the cost of multiplying univariate polynomials of degree less than b, see [34, Chap. 8]. Using the estimate $\mathsf{M}(b) \in O(b\log(b)\log\log(b))$ from [12], this respectively gives the bounds

$$O(d_{n+1}^2 B \log(B)) \quad \text{and} \quad O(d_{n+1}^2 B \log^2(B) \log\log(B)).$$

These estimates are far from optimal. A first important improvement consists in interpolating in the first place only the *leading coefficients* of the subresultants, and

recover all other coefficients when needed. This is sufficient for the algorithm of Section 3.4. This idea brings the following result.

Lemma 15. *Constructing the SCube can be done within*

$$O(d_{n+1}^2 B + d_{n+1} B \log^2(B) \log\log(B))$$

operations in $\mathbf{k}$. *If multi-dimensional FFT can be used then this estimate becomes* $O(d_{n+1}^2 B + d_{n+1} B \log(B))$ *operations in* $\mathbf{k}$.

Another desirable improvement would consist in using fast arithmetic based on *Half-GCD* techniques [34], with the goal of reducing the total cost to $O^{\sim}(d_{n+1}B)$, which is the best known bound for computing the resultant, or a given subresultant. However, as of now, we do not have such a result, due to the possible splittings.

We return now to the question of solving two equations. Our goal is to estimate the cost of computing the polynomials R and G in the context of Formula 3.1 p. 39. We propose an approach where the computation of G essentially comes for free, once R has been computed. This is a substantial improvement compared to traditional methods, such as [41, 65], which compute G without recycling the intermediate calculations of R. With the above assumptions and notations, we saw that the resultant R can be computed in at most $O(d_{n+1}B\log(B) + d_{n+1}^2 B)$ operations in $\mathbf{k}$. In many cases (typically, with random systems), G has degree one in $y = x_{n+1}$. Then, the GCD G can be computed within the same bound as the resultant. Besides, in this case, one can use the Half-GCD approach instead of computing all subresultants of P and Q. This leads to the following result in the bivariate case; we omit its proof here.

Corollary 4. *With* $n = 1$, *assuming that* $V(H, P, Q)$ *is empty, and assuming* $\mathsf{deg}(G, y) = 1$, *solving the input system* $P = Q = 0$ *can be done in* $O^{\sim}(d_2^2 d_1)$ *operations in* $\mathbf{k}$.

4.3 Regularity test in dimension zero

The operation Regularize specified in Section 3.2.1 of Chapter 3 is a core routine in methods computing triangular decompositions. It has been used in the algorithm RGSZR presented in Section 3.4 of Chapter 3. Algorithms for this operation appear in [41, 65].

The purpose of this section is to show how to realize efficiently this operation. For simplicity, we restrict ourselves to regular chains with zero-dimensional saturated

ideals, in which case the separate operation of [41] and the regularize operation [65] are similar. We also restrict ourselves to reduced and normalized regular chains, which implies that these regular chains are reduced lexicographical Gröbner bases.

For such a regular chain T in $\mathbf{k}[\mathbf{x}]$ and a polynomial $p \in \mathbf{k}[\mathbf{x}]$, we denote by RegularizeDim0(p, T) the function call Regularize(p, T). In broad terms, it "separates" the points of $V(T)$ that cancel p from those which do not. The output is a set of regular chains $\{T_1, \ldots, T_e\}$ such that the points of $V(T)$ which cancel p are given by the T_i's modulo which p is null.

Algorithm 4 differs from those with similar specification in [41, 65] by the fact that it creates opportunities for using modular methods and fast polynomial arithmetic. Our first trick is based on the following result (Theorem 1 in [14]): the polynomial p is invertible modulo T if and only if the iterated resultant of p with respect to T is non-zero. The correctness of Algorithm 4 follows from this result, the specification of the operation RegularGcd and an inductive process. Similar proofs appear in [41, 65]. A complexity analysis of Algorithm 4, under some genericity assumptions, is reported at the end of this section.

The main novelty of Algorithm 4 is to employ the fast evaluation/interpolation strategy described in Section 4.2. In our implementation of Algorithm 4, at Line **6**, we compute the SCube representing the subresultant chain of q and C_v. This allows us to compute the resultant r and then to compute the regular GCDs (g, E) at Line **14** from the same SCube. In this way, intermediate computations are recycled. Moreover, fast polynomial arithmetic is involved through the manipulation of the SCube.

In Algorithm 4, a routine RegularizeInitDim0 is called, whose specification is given below. See [65] for an algorithm. Briefly speaking, this routine splits a regular chain T into regular chains $T_1, \ldots, T_e$ according to a polynomial p such that for each $i = 1 \cdots e$ the polynomial p reduces modulo sat(T_i) to a constant polynomial or to a polynomial with a regular initial.

4.4 Complexity analysis

We shall now estimate the running time of Algorithm 4 under the following two genericity assumptions.

($\mathbf{H_1}$) T generates a radical ideal,

Algorithm 4: Regularize a polynomial in dimension zero

RegularizeDim0(p, T)

Input : T normalized reduced zero-dimensional regular chain and p polynomial, both in $\mathbf{k}[x_1, \ldots, x_n]$, with p reduced w.r.t. T.

Output : see the specification in Section 3.2.2.

```
1  Results ← ∅
2  for (q, C) ∈ RegularizeInitDim0(P, T) do
3      if q ∈ k then Results ← {C} ∪ Results
4      else
5          v ← mvar(q)
6          r ← res(q, C_v, v)
7          r ← NormalForm(r, C_<v)
8          for D ∈ RegularizeDim0(r, C_<v) do
9              s ← NormalForm(r, D)
10             if s ≠ 0 then
11                 U ← {D ∪ {C_v} ∪ C_>v}
12                 Results ← {C} ∪ Results
13             else
14                 for (g, E) ∈ RegularGcd(q, C_v, D) do
15                     g ← NormalForm(g, E)
16                     U ← {E ∪ {g} ∪ D_>v}
17                     Results ← {C} ∪ Results
18                     c ← NormalForm(quo(C_v, g), E)
19                     if deg(c, v) > 0 then
20                         Results ←
                           RegularizeDim0(q, E ∪ c ∪ C_>v) ∪ Results
21 return Results
```

($\mathbf{H_2}$) none of the calls to RegularizeDim0 splits its second argument into several regular chains.

Ensuring that Hypothesis ($\mathbf{H_1}$) holds is standard. This is done by adapting the squarefree part computation of univariate polynomial with coefficients in a field to coefficients in products of fields, see [65]. Hypothesis ($\mathbf{H_1}$) holds if T generates a maximal ideal. It is also likely to hold on a random dense input, as observed in our experimentation. Analyzing the running time of Algorithm 4 without Hypothesis ($\mathbf{H_2}$) leads to additional difficulties which can be handled using the techniques of [22].

In order to proceed with our analysis, we need some notations. We define $\mathrm{logp}(x) = \log_2(\max(2, x))$ and $\mathrm{llogp}(x) = \mathrm{logp}(\mathrm{logp}(x))$ for any real value x. Ob-

Algorithm 5: Regularize the initial of a polynomial in dimension zero

RegularizeInitDim0(p, T)

Input : T a normalized zero-dimensional regular chain and p a polynomial, both in $\mathbf{k}[x_1, \ldots, x_n]$

Output : A set of pairs $\{(p_i, T_i) \mid i = 1 \cdots e\}$, in which p_i is a polynomial and T_i is a regular chain, such that either p_i is a constant or its initial is regular modulo $\mathsf{sat}(T_i)$, $p \equiv p_i \bmod \mathsf{sat}(T_i)$ holds, and we have $T \longrightarrow (T_1, \ldots, T_e)$.

```
p ← NormalForm(p, T)
Tasks ← {(p, T)}
Results ← ∅
while Tasks ≠ ∅ do
    Take a pair (q, C) out of Tasks
    if q ∈ k then
        Results ← {(q, C)} ∪ Results
    else
        for D ∈ RegularizeDim0(init(q), C) do
            t ← NormalForm(tail(q), D)
            h ← NormalForm(init(q), D)
            if h ≠ 0 then
                Results ← {(h rank(q) + t, D)} ∪ Results
            else
                Tasks ← {(t, D)} ∪ Tasks
return Results
```

serve that for all a, b we have $\mathrm{logp}(ab) \leq \mathrm{logp}(a)\mathrm{logp}(b)$. Let d_i be the degree in x_i of the polynomial T_{x_i}. Let $s_1, \ldots, s_n$ be positive integers and let $s \in \mathbf{k}[x_1, \ldots, x_n]$ be a polynomial satisfying $\mathsf{deg}(s, x_i) < s_i$ for all $i = 1 \cdots n$. We denote by $\mathsf{NF}(s_1, \ldots, s_n, d_1, \ldots, d_n)$ an upper bound for the number of operations in $\mathbf{k}$ performed when computing the normal form of s w.r.t. T. If no confusion is possible, we simply write $\mathsf{NF}(s_1, \ldots, s_n)$ instead of $\mathsf{NF}(s_1, \ldots, s_n, d_1, \ldots, d_n)$. Next, we denote by $\mathsf{RZ}(d_1, \ldots, d_n)$ (resp. $\mathsf{M}_T(d_1, \ldots, d_n)$) an upper bound for the number of operations in $\mathbf{k}$ performed when computing RegularizeDim0(p, T) where p is reduced w.r.t. T (resp. when multiplying modulo $\langle T \rangle$ two polynomials reduced w.r.t. T). In [52] it is shown that there exists a constant $\mathsf{K} > 1$ such that

$$\mathsf{M}_T(d_1, \ldots, d_n) \leq 4^n \, \mathsf{K} \, D_n \, \mathrm{logp}(D_n)\mathrm{llogp}(D_n)$$

holds where $D_k = d_1 \cdots d_k$. By convention $D_0 = 1$.

Lemma 16. *With the above notations, we have:*

$$\mathsf{NF}(s_1, \ldots, s_n) \leq 5\,\mathsf{K}\,\mathrm{logp}(\sigma)\,\mathrm{llogp}(\sigma)\,\mathrm{logp}(D_{n-1})\,\mathrm{llogp}(D_{n-1}) \sum_{i=1}^{n} 4^{i-1} S_i D_{i-1},$$

where we define $\sigma = \max(s_1, \ldots, s_n)$ *and* $S_i = s_i \cdots s_n$, *for all* $i = 1 \cdots n$.

Proof. Let $c_0, \ldots, c_t$ be the coefficients of s w.r.t. x_n such that s writes $\sum_{i=0}^{t} c_i x_n^i$. To compute $\mathsf{NormalForm}(s, T)$ we start by computing s' which is $\sum_{i=0}^{t} c'_i x_n^i$ where c'_i is $\mathsf{NormalForm}(c_i, T_{<x_n})$. Since $t < s_n$, this first step costs at most $s_n \mathsf{NF}(s_1, \ldots, s_{n-1})$ operations in $\mathbf{k}$. Then, we compute the remainder in the Euclidean division of s' by T_{x_n} modulo $\langle T_{<x_n} \rangle$. Using the results of Chapter 9 in [34], this latter step amounts to at most $5\mathsf{M}(s_n)\mathsf{M}_T(d_1, \ldots, d_{n-1})$. This leads to the following inequality

$$\mathsf{NF}(s_1, \ldots, s_n) \leq s_n \mathsf{NF}(s_1, \ldots, s_{n-1}) + 5\mathsf{M}(s_n)\mathsf{M}_T(d_1, \ldots, d_{n-1}).$$

Unrolling this relation yields

$$\mathsf{NF}(s_1, \ldots, s_n) \leq 5 s_n \cdots s_2 \mathsf{M}(s_1) + \sum_{i=2}^{n} 5 s_n \cdots s_{i+1} \mathsf{M}(s_i)\mathsf{M}_T(d_1, \ldots, d_{i-1}).$$

Therefore, we have

$$\mathsf{NF}(s_1, \ldots, s_n) \leq 5\mathsf{K}\mathrm{logp}(\sigma)\,\mathrm{llogp}(\sigma)\,\mathrm{logp}(D_{n-1})\,\mathrm{llogp}(D_{n-1}) \sum_{i=1}^{n} 4^{i-1} S_i D_{i-1}.$$

□

Lemma 17. *With notations of Lemma 16, assuming* $d_{n+1} \geq 2$ *and* $s_i = 2 d_i d_{n+1}$ *for all* $i = 1 \cdots n$, *we have*

$$\mathsf{NF}(s_1, \ldots, s_n) \leq 80\,\mathsf{K}\,n\,2^n\,d_{n+1}^n\,D_n\,\mathrm{logp}^2(D_n)\,\mathrm{llogp}^2(D_n).$$

Proof. We apply Lemma 16. First, we observe that $\mathrm{logp}(\sigma) \leq 2\,\mathrm{logp}(D_n)$ holds if $n > 1$. However, to cover $n = 1$, we use the estimate $\mathrm{logp}(\sigma) \leq 4\,\mathrm{logp}(D_n)$. Since $\mathrm{logp}(D_n) \geq 1$ holds, we deduce $\mathrm{llogp}(\sigma) \leq 4\,\mathrm{llogp}(D_n)$. Next, we observe that

$S_i = (2d_{n+1})^{n-i+1} d_i \cdots d_n$ holds which brings

$$\sum_{i=1}^{n} 4^{i-1} S_i D_{i-1} = 2^n d_{n+1}^n D_n \sum_{i=1}^{n} (2/d_{n+1})^{i-1} \leq n\, 2^n\, d_{n+1}^n\, D_n$$

and the conclusion follows. □

As in Section 4.2 we consider two polynomials $P, Q \in [x_1, \ldots, x_n, x_{n+1}]$ with positive degree in $y = x_{n+1}$ such that we have $0 < \mathsf{deg}(Q, x_{n+1}) \leq \mathsf{deg}(P, x_{n+1}) =: d_{n+1}$. We assume that the initials of P, Q are regular w.r.t. $\mathsf{sat}(T)$, that the resultant of P, Q w.r.t. x_{n+1} belongs to $\mathsf{sat}(T)$ and that all coefficients of P and Q w.r.t. x_{n+1} are reduced w.r.t. T. Let us denote by $\mathsf{SRC}(d_1, \ldots, d_n, d_{n+1})$ an upper bound for the number of operations in $\mathbf{k}$ necessary to construct the SCube of P, Q. It follows from Lemma 15 that there exists a constant $\mathsf{C} > 0$ such that

$$\mathsf{SRC}(d_1, \ldots, d_n, d_{n+1}) \leq \mathsf{C}\left(d_{n+1}^2 B_n + d_{n+1} B_n \mathrm{logp}^2(B_n) \mathrm{llogp}(B_n)\right) \tag{4.1}$$

where $B_n = 2^n d_{n+1}^n D_n$. Moreover, one can choose C such that each coefficient w.r.t. x_{n+1} of a subresultant of P and Q w.r.t. x_{n+1} can be interpolated within $\mathsf{C}\, B_n \mathrm{logp}^2(B_n) \mathrm{llogp}(B_n)$ operations in $\mathbf{k}$.

We denote by $\mathsf{GCD}(d_1, \ldots, d_n, d_{n+1})$ an upper bound for the number of operations in $\mathbf{k}$ performed when computing a regular GCD sequence of P, Q modulo $\mathsf{sat}(T)$. We have the following result.

Lemma 18. *Under Hypotheses* $(\mathbf{H_1})$ *and* $(\mathbf{H_2})$*, we have:*

$$\begin{aligned} \mathsf{GCD}(d_1, \ldots, d_{n+1}) \leq\ & \mathsf{SRC}(d_1, \ldots, d_n, d_{n+1}) + \mathsf{RZ}(d_1, \ldots, d_n) + \\ & \tfrac{d_{n+1}(d_{n+1}+1)}{2} \left(\mathsf{C}\, B_n \mathrm{logp}^2(B_n) \mathrm{llogp}(B_n) + \mathsf{NF}(s_1, \ldots, s_n)\right) \end{aligned}$$

where $s_i = 2 d_i d_{n+1}$ *for all* $i = 1 \cdots n$.

Proof. Recall that Hypothesis $(\mathbf{H_2})$ means that computations do not split. This implies that when Algorithm 3 calls $\mathsf{RegularizeDim0}(c_i, C)$ with a (reduced, normalized, zero-dimensional) regular chain C and with a polynomial c_i reduced w.r.t. C, then c_i is either null or invertible modulo $\langle C \rangle$. Consider now the *candidate search phase* (Lines **4** to **15**) in Algorithm 3. With the notations of this algorithm, consider an item $[i, C]$ at Line **5**. If c_i belongs to $\mathsf{sat}(C)$ then the whole subresultant S_i belongs to $\mathsf{sat}(C)$. This follows from Lemma 12 and the fact that $\mathsf{sat}(C)$ is radical (Hypothesis $(\mathbf{H_2})$). If c_i does not belong to $\mathsf{sat}(C)$ then c_i is invertible modulo $\mathsf{sat}(C)$ and thus

S_i is a candidate. This implies that, in the worst case, the *candidate search phase* is accomplished by:

- interpolating all subresultant of P and Q w.r.t. x_{n+1} from the SCube,
- computing the normal form of all these coefficients w.r.t. T,
- performing one regularity test of a polynomial which is not in $\langle T \rangle$.

Finally, Hypothesis $(\mathbf{H_1})$ together with Lemma 14 implies that the candidate is actually a regular GCD of P, Q modulo $\mathsf{sat}(T)$. Hence the *candidate check phase* of Algorithm 3 comes at no cost. The conclusion follows. □

Lemma 19. *Under Hypotheses* $(\mathbf{H_1})$ *and* $(\mathbf{H_2})$ *and assuming* $d_{n+1} \geq 2$*, we have*

$$\mathsf{GCD}(d_1, \ldots, d_{n+1}) \leq O(n^2\, 2^n\, d_{n+1}^{n+2}\, D_n\, L_n) + \mathsf{RZ}(d_1, \ldots, d_n),$$

where $L_n = \mathrm{logp}(n)\, \mathrm{logp}^2(d_{n+1})\, \mathrm{llogp}(d_{n+1})\, \mathrm{logp}^2(D_n)\, \mathrm{llogp}^2(D_n)$.

Proof. From Lemma 18 and Equation (4.1) we have

$$\begin{array}{rl} \mathsf{GCD}(d_1, \ldots, d_n, d_{n+1}) \leq & \mathsf{C}\, \frac{3(d_{n+1}+1)d_{n+1}}{2} B_n \mathrm{logp}^2(B_n) \mathrm{llogp}(B_n) \\ & \frac{(d_{n+1}+1)d_{n+1}}{2} \mathsf{NF}(s_1, \ldots, s_n) + \mathsf{RZ}(d_1, \ldots, d_n) \end{array} \tag{4.2}$$

We shall simplify the above inequality. Since $B_n = 2^n d_{n+1}^n D_n$ and $n \geq 1$, we deduce

$$\mathrm{logp}(B_n) \;\leq\; n + n\log_2(d_{n+1}) + \log_2(D_n) \;\leq\; 3\, n\, \mathrm{logp}(d_{n+1})\, \mathrm{logp}(D_n),$$

and

$$\mathrm{llogp}(B_n) \leq 2\, \mathrm{logp}(n)\, \mathrm{llogp}(d_{n+1})\, \mathrm{llogp}(D_n),$$

which leads to

$$B_n\, \mathrm{logp}^2(B_n)\, \mathrm{llogp}(B_n) \leq 18\, n^2\, 2^n\, d_{n+1}^n\, D_n\, L_n. \tag{4.3}$$

Next, we deduce from Lemma 17 that

$$\mathsf{NF}(s_1, \ldots, s_n) \leq 80\, \mathsf{K}\, n\, 2^n\, d_{n+1}^n\, D_n\, L_n \tag{4.4}$$

Using $\frac{1}{2}(d_{n+1}+1)d_{n+1} \leq d_{n+1}^2$ together with (4.2), (4.3) and (4.4) we obtain

$$\mathsf{GCD}(d_1, \ldots, d_n, d_{n+1}) \;\leq\; (54\, \mathsf{C}\, n + 80\, \mathsf{K})\; n\, 2^n\, d_{n+1}^{n+2}\, D_n\, L_n + \mathsf{RZ}(d_1, \ldots, d_n).$$

This completes the proof. □

Theorem 6. *Under Hypotheses* $(\mathbf{H_1})$ *and* $(\mathbf{H_2})$ *and assuming* $d_i \geq 2$ *for all* $i = 1 \cdots n$ *we have, for* $n \geq 2$

$$\mathsf{RZ}(d_1, \ldots, d_n) \leq O(n^2\, 2^{n-1})\, d_n^{n+1}\, D_{n-1}\, L_{n-1} + 2\, \mathsf{RZ}(d_1, \ldots, d_{n-1}) \tag{4.5}$$

which implies

$$\mathsf{RZ}(d_1, \ldots, d_n) \in O^{\sim}(2^n) \sum_{i=2}^{n} \left(i^2\, d_i^i\, D_i\right). \tag{4.6}$$

Proof. We follow Algorithm 4, which computes $\mathsf{RegularizeDim0}(p, T)$. Recall that the input polynomial p is reduced w.r.t T. Since we are looking for an upper bound for $\mathsf{RZ}(d_1, \ldots, d_n)$ we can assume that the main variable of p is x_n. Hypothesis $(\mathbf{H_2})$ implies that $\mathsf{init}(p)$ is invertible modulo $\langle T \rangle$ and thus that executing Line **2** amounts at most to $\mathsf{RZ}(d_1, \ldots, d_{n-1})$. Observe that at Line **5** we have $q = p$ and $C_v = T_{x_n}$. The next cost is at Lines **6** and **7** with the computation of the SCube of p and T_{x_n} w.r.t. x_n, the interpolation of their resultant r and the computation of $\mathsf{NormalForm}(r, T_{<x_n})$. We observe that this cost is included in (resp. dominated by) the estimate of $\mathsf{GCD}(d_1, \ldots, d_{n-1}, d_n)$ given by Lemma 18, if $\mathsf{NormalForm}(r, T_{<x_n}) = 0$ (resp. r is invertible modulo $\langle T_{<x_n} \rangle$). The next cost is at Line **8** with the call $\mathsf{RegularizeDim0}(r, T_{<x_n})$, amounting at most to $\mathsf{RZ}(d_1, \ldots, d_{n-1})$. Hypothesis $(\mathbf{H_2})$ implies that Line **9** comes at no cost. At this point either $r \notin \mathsf{sat}(T_{<x_n})$ holds and the algorithm terminates, or the next expense is at Line **14** which fits within $\mathsf{GCD}(d_1, \ldots, d_{n-1}, d_n)$. In this latter case, $(\mathbf{H_2})$ implies $\mathsf{deg}(g, x_n) = \mathsf{deg}(q, x_n)$ and no other computations take place. Finally, we obtain Relation (4.5) by virtue of Lemma 19. □

Corollary 5. *Under Hypotheses* $(\mathbf{H_1})$ *and* $(\mathbf{H_2})$ *and assuming* $d_i \geq 2$ *for all* $i = 1 \cdots n$ *we have, for* $n \geq 2$

$$\mathsf{GCD}(d_1, \ldots, d_n, d_{n+1}) \in O^{\sim}(n^2\, 2^n) d_{n+1}^{n+2} D_n + O^{\sim}(2^n) \sum_{i=2}^{n} \left(i^2\, d_i^i\, D_i\right). \tag{4.7}$$

Proof. The claim follows from Theorem 6 and Lemma 19. □

Essentially, Relation (4.7) depends "quadratically" on the product of the degrees $d_1, \ldots, d_n, d_{n+1}$. This is clear when $d_1 = \cdots = d_n = d_{n+1}$ holds. Moreover the "exponential factor" is only 2^n. In [22], the authors provide an algorithm with the same specification as Algorithm 4. Under the same hypotheses, they achieve a running estimate which depends "linearly" (up to logarithmic factors) on the product of the degrees $d_1, \ldots, d_n, d_{n+1}$. However, their "exponential factor" is of the form c^n where

$\mathsf{c} \geq 700$. Since practical values for $d_1, \ldots, d_n, d_{n+1}$ are often below the hundreds, in particular for d_{n+1} with n large, this suggests that the algorithms presented in this chapter are probably more suitable for implementation than those of [22].

4.5 Experimentation

We have implemented in the C language all the algorithms presented in the previous sections. The corresponding functions rely on the asymptotically fast arithmetic operations from our `modpn` library [50]. For this new code, we have also realized a MAPLE interface, called FastArithmeticTools, which is a new module of the `RegularChains` library [48].

In this section, we compare the performance of our `FastArithmeticTools` commands with MAPLE's and MAGMA's existing counterparts. For MAPLE, we use its latest release, namely version 13; For MAGMA we use Version `V2.15-4`, which is the latest one at the time of writing this chapter. However, for this release, the MAGMA commands `TriangularDecomposition` and `Saturation` appear to be some time much slower than in Version `V2.14-8`. When this happens, we provide timings for both versions.

We have three test cases dealing respectively with the solving of bivariate systems, the solving of two-equation systems and the regularity testing of a polynomial w.r.t. a zero-dimensional regular chain. In our experimentation all polynomial coefficients are in a prime field whose characteristic is a 30-bit prime number. For each of our figure or table the degree is the total degree of any polynomial in the input system. All the benchmarks were conducted on a 64-bit Intel Pentium VI Quad CPU 2.40 GHZ machine with 4 MB L2 cache and 3 GB main memory.

For solving bivariate systems we compare the command `Triangularize` to the command `BivariateModularTriangularize` of the module `FastArithmeticTools`. Indeed both commands have the same specification for such input systems. Note that `Triangularize` is a high-level generic code which applies to any type of input system and which does not rely on fast polynomial arithmetic or modular methods. On the contrary, `BivariateModularTriangularize` is specialized to bivariate systems (see Corollary 4 in Section 4.2) is mainly implemented in C and is supported by the `modpn` library. `BivariateModularTriangularize` is an instance of a more general fast algorithm called `FastTriangularize`; we use this second name in our figures.

Since a triangular decomposition can be regarded as a "factored" lexicographic Gröbner basis we also benchmark the computation of such bases in

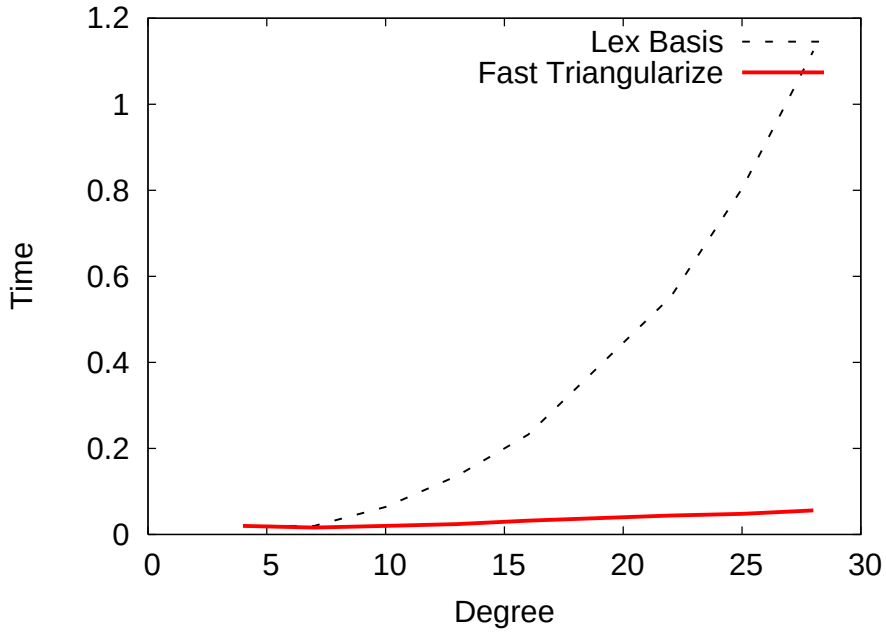

Figure 4.1: Timing for bivariate generic dense systems

MAPLE and MAGMA. Figure 4.1 compares `FastTriangularize` and (lexicographic) `Groebner:-Basis` in MAPLE on generic dense input systems. On the largest input example the former solver is about 20 times faster than the latter.

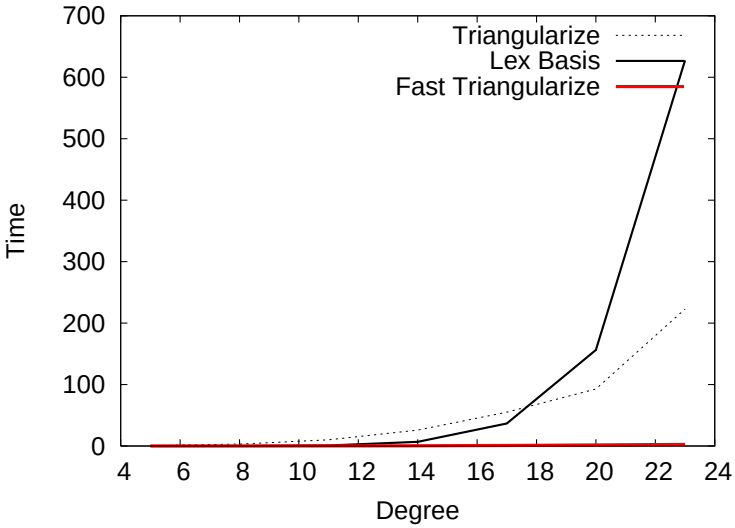

Figure 4.2: Highly non-equiprojectable bivariate systems

Figure 4.2 compares `FastTriangularize` and (lexicographic) `Groebner:-Basis` on highly non-equiprojectable dense input systems; for these systems the number

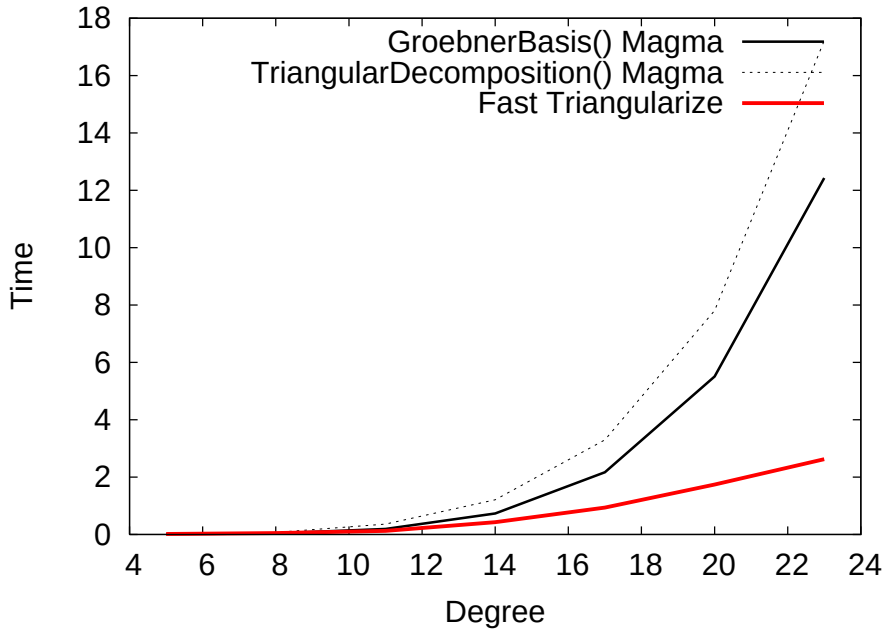

Figure 4.3: Highly non-equiprojectable systems

of equiprojectable components is about half the degree of the variety. At the total degree 23 our solver is approximately 100 times faster than `Groebner:-Basis`.

Figure 4.3 compares `FastTriangularize`, `GroebnerBasis` in MAGMA and `TriangularDecomposition` in MAGMA on the same set of highly non-equiprojectable dense input systems. Once again our solver outperforms its competitors.

For solving systems with two equations, we compare `FastTriangularize` with `GroebnerBasis` in MAGMA. On Figure 4.4 these two solvers are simply referred as MAGMA and MAPLE. For this benchmark the input are generic dense trivariate systems.

Figures 4.5, 4.6 and 4.7 compare our fast regularity test algorithm (Algorithm 4) with the `RegularChains` library `Regularize` and its MAGMA counterpart. More precisely, in MAGMA, we first saturate the ideal generated by the input zero-dimensional regular chain T with the input polynomial P using the `Saturation` command. Then the `TriangularDecomposition` command decomposes the output produced by the first step. The total degree of the input i-th polynomial in T is d_i.

For Figure 4.5 and Figure 4.6, the input T and P are randomly generated such that the intermediate computations do not split. In this non-splitting cases, our fast `Regularize` algorithm is significantly faster than the other commands. For Figure 4.7, the input T and P are constructed such that many intermediate computations need to split. In this case, our fast `Regularize` algorithm is slightly slower than its MAGMA

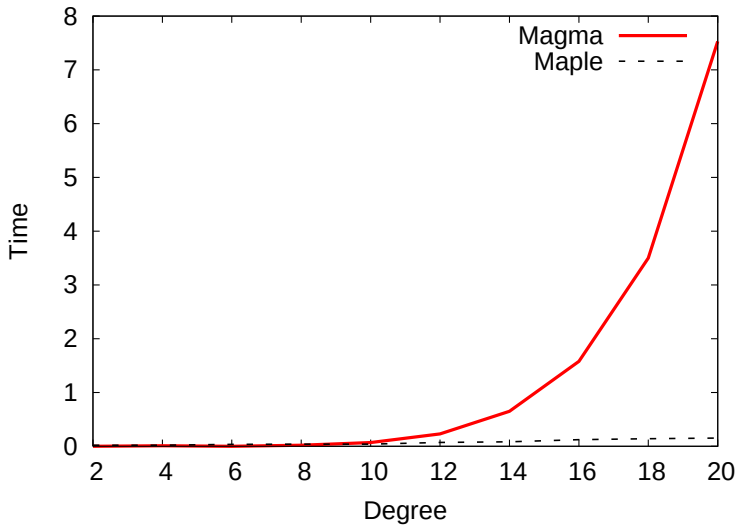

Figure 4.4: Generic dense trivariate systems

d_1	d_2	Reg.	Fast Reg.	Magma	d_1	d_2	Reg.	Fast Reg.	Magma
2	2	0.052	0.016	0.000	20	38	69.876	0.776	3.660
4	6	0.236	0.016	0.010	22	42	107.154	0.656	6.600
6	10	0.760	0.016	0.010	24	46	156.373	1.036	10.460
8	14	1.968	0.020	0.050	26	50	220.653	2.172	17.110
10	18	4.420	0.052	0.090	28	54	309.271	1.640	25.900
12	22	8.784	0.072	0.220	30	58	434.343	2.008	42.600
14	26	15.989	0.144	0.500	32	62	574.923	4.156	57.000
16	30	27.497	0.208	0.990	34	66	746.818	6.456	104.780
18	34	44.594	0.368	1.890					

Figure 4.5: bivariate random dense

counterpart, but still much faster than the generic (non-modular and non-supported by `modpn`) `Regularize` command of the `RegularChains` library. The slow down w.r.t.

d_1	d_2	d_3	Reg.	Fast Reg.	Magma	d_1	d_2	d_3	Reg.	Fast Reg.	Magma
2	2	3	0.240	0.008	0.000	8	14	21	168.910	2.204	8.250
3	4	6	1.196	0.020	0.020	9	16	24	332.036	14.764	23.160
4	6	9	4.424	0.032	0.030	10	18	27	>1000	21.853	61.560
5	8	12	12.956	0.148	0.200	11	20	30	>1000	57.203	132.240
6	10	15	33.614	0.360	0.710	12	22	33	>1000	102.830	284.420
7	12	18	82.393	1.108	2.920						

Figure 4.6: trivariate random dense

d_1	d_2	d_3	Reg.	Fast Reg.	V2.15-4	V2.14-8
2	2	3	0.184	0.028	0.000	0.000
3	4	6	0.972	0.060	0.000	0.010
4	6	9	3.212	0.092	>1000	0.030
5	8	12	8.228	0.208	>1000	0.150
6	10	15	21.461	0.888	807.850	0.370
7	12	18	51.751	3.836	>1000	1.790
8	14	21	106.722	9.604	>1000	2.890
9	16	24	207.752	39.590	>1000	10.950
10	18	27	388.356	72.548	>1000	19.180
11	20	30	703.123	138.924	>1000	56.850
12	22	33	>1000	295.374	>1000	76.340

Figure 4.7: trivariate dense with many splittings

the MAGMA code is due to the (large) overheads of the C - MAPLE interface, see [50] for details.

4.6 Summary

In this chapter, we report our C implementation of subresultant chain construction and regular GCD algorithm. The running time estimates of Section 4.3 suggests that the algorithms presented in this chapter are more suitable for implementation than those of [22]. Our experimental results of Section 4.5 illustrate the high efficiency of our algorithms.

Chapter 5

Modular FFT on GPUs

5.1 Introduction

Polynomials and matrices are the fundamental objects on which most computer algebra algorithms operate. In the past 15 years, significant efforts have been deployed by different groups of researchers for delivering highly efficient software packages for computing symbolically with polynomials and matrices, like LINBOX, MAGMA, and NTL [54, 1, 77]. However, most of these works are dedicated to serial implementations, in particular in the case of polynomials. Only a few studies [61, 68, 69] report on parallel implementation (targeting multicores) of polynomial arithmetic. None of the computer algebra software packages available today takes advantage of graphics processing units (GPUs) in support of libraries for polynomial arithmetic. The work reported in this chapter aims at filling this gap.

This contrasts sharply with the state of affairs in numerical linear algebra and in digital signal processing. For instance, the commercialized software system MATLAB with its Parallel Computing Toolbox [56] and GPU Toolbox [37] provides programming support for different parallelism paradigms (data-parallelism, MPI, multithreading) and parallel architectures (GPUs, multicores, clusters) together with many library functions taking advantage of this support. In digital signal processing, in particular for the computation of Fast Fourier Transforms (FFTs), the use of hardware acceleration technologies, notably GPUs, has been investigated in several works [33, 63, 36, 70].

In this chapter, we present a GPU implementation of fast polynomial multiplication. We focus on dense univariate polynomials over prime fields for the following reasons. First, many algorithms in symbolic computation tend to densify intermediate data, even if the input and output are sparse. Second, multivariate polynomial

multiplication can be reduced to univariate multiplication through the so-called Kronecker's substitution. Third, computation with polynomials over non-prime fields can be reduced to the prime field case by means of modular techniques. We refer to the landmark book *Modern Computer Algebra* [34] for an extensive presentation of these ideas.

This reduction to dense univariate polynomials over coefficient fields $\mathbb{Z}/p\mathbb{Z}$, where p is a prime, allows us to rely on FFT techniques, which is the basis of fast polynomial arithmetic [12]. However, as detailed in Section 5.2.3, FFT computations over finite fields present specific challenges. For this reason, techniques for FFTs with floating point number coefficients are not sufficient for supporting polynomial multiplication over finite fields. This motivates the work reported in this chapter.

Most serial implementations of FFT over finite fields, see [31] and the references therein, rely on the radix-2 Cooley-Tukey Formula [19]. On multicores, the row-column FFT algorithm is used successfully, see [68, 69]. In the case of GPUs, to which this chapter is devoted, it is natural to revisit the popular FFT formulas of Cooley-Tukey and Stockham [78] in the context of finite fields. We review these formulas in Section 5.2.2. As in [73, 13] we take advantage of the formalism of tensorial calculus to generate code and identify our GPU kernels. The Cooley-Tukey and Stockham formulas differ only in the way that intermediate computations are stored. We concentrated our efforts on these two formulas, despite of the existence of other formulas for computing FFTs, for the following reasons. First, the radix-2 Cooley-Tukey formula is well understood in the context of finite fields. Second, in numerical computing, the Stockham formula seems to be well-suited for GPU implementation [33].

In this work, we present our detailed implementations of the Cooley-Tukey and Stockham FFT formulas, aiming at utilizing the horsepower of Graphics Processing Units (GPUs). The organization of the chapter is as follows. In Section 5.2, we first formalize FFTs in terms of Kronecker (tensor) product, then we discuss an efficiency-critical operation in a finite field, namely modular multiplication. Sections 5.4 and Section 5.5 focus on our CUDA [2] implementation of the Cooley-Tukey and Stockham FFTs. We present experimental results in Section 5.6 and draw conclusions in the end.

This joint work with Marc Moreno Maza is reported in [57].

5.2 Preliminaries

This section reviews the Fast Fourier Transform (FFT) in the language of tensorial calculus, see [55] for an extensive presentation. This formalism facilitates code generation as explained in [13, 32], and in particular it helps identifying GPU kernel specifications. We also highlight the specific features of FFTs over finite fields and refer to [69] for details. Throughout this chapter, we denote by $\mathbf{k}$ a field. In practice, this field is often a prime field $\mathbb{Z}/p\mathbb{Z}$ where p is a prime number greater than 2.

5.2.1 Basic operations on matrices

Definition 6. *Let n, m, q, s be positive integers and let A, B be two matrices over $\mathbf{k}$ with respective formats $m \times n$ and $q \times s$. The tensor (or Kronecker) product of A by B is an $mq \times ns$ matrix over $\mathbf{k}$ denoted by $A \otimes B$ and defined by*

$$A \otimes B = [a_{k\ell}B]_{k,\ell} \quad \textit{with} \quad A = [a_{k\ell}]_{k,\ell} \tag{5.1}$$

Example 7. *Let A and B be 2×2 matrices,*

$$A = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix} \quad \textit{and} \quad B = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}.$$

Then their tensor products are

$$A \otimes B = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 2 & 2 & 3 & 3 \\ 2 & 2 & 3 & 3 \end{bmatrix} \quad \textit{and} \quad B \otimes A = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 3 & 2 & 3 \\ 0 & 1 & 0 & 1 \\ 2 & 3 & 2 & 3 \end{bmatrix}.$$

We summarize several properties of tensor product as follows:

Proposition 8. *Assume A, B, C, and D are matrices of suitable sizes. Then we have*

(1) $A \otimes (B + C) = A \otimes B + A \otimes C$,

(2) $(B + C) \otimes A = B \otimes A + C \otimes A$,

(3) $(\lambda\, A) \otimes B = A \otimes (\lambda\, B) = \lambda(A \otimes B)$, *with λ being a scalar,*

(4) $(A \otimes B) \otimes C = A \otimes (B \otimes C) = A \otimes B \otimes C$,

(5) $(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$, *the mixed-product property.*

The first three properties say that the tensor product operator is bilinear, that is, linear for both operands. The fourth property says that the operator $\otimes$ is associative.

Denoting by I_n the identity matrix of order n, we emphasize two particular types of tensor products, $I_n \otimes A_m$ and $A_n \otimes I_m$, where A_m (resp. A_n) is a square matrix of order m (resp, n) over $\mathbf{k}$. The following pseudo C-code implements the endomorphism (linear map) of $\mathbf{k}^{nm}$ defined $\mathbf{x} \longmapsto \mathbf{y} := (I_n \otimes A_m)\mathbf{x}$:

```
for (k = 0; k < n; ++k) {
    for (i = 0; i < m; ++i) {
        t = 0;
        for (j = 0; j < m; ++j) {
            t += A[i][j] * x[k*m+j];
        }
        y[k*q+i] = t;
    }
}
```

The cost of above code is $\Theta(m^2 n)$, while computing the product of an $mn \times mn$ matrix by a vector uses $\Theta(m^2 n^2)$ arithmetic operations. This type of saving comes from the utilization of special structures in the tensor products. This can be viewed as an opportunity for block-level parallelism as illustrated by the example below:

$$I_4 \otimes \mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 & & & & & & \\ 1 & -1 & & & & & & \\ & & 1 & 1 & & & & \\ & & 1 & -1 & & & & \\ & & & & 1 & 1 & & \\ & & & & 1 & -1 & & \\ & & & & & & 1 & 1 \\ & & & & & & 1 & -1 \end{bmatrix}$$

in which blocks share the same computations while on different sub-vectors.

Whereas the pseudo C-code below implements the endomorphism of $\mathbf{k}^{nm}$ defined by $\mathbf{x} \longmapsto \mathbf{y} := (A_n \otimes I_m)\mathbf{x}$:

```
for (k = 0; k < n; ++k) {
    for (i = 0; i < m; ++i) {
```

```
        t = 0;
        for (j = 0; j < n; ++j) {
            t += A[k][j] * x[j*m+i]
        }
        y[k*q+i] = t;
    }
}
```

The cost of above code is also $\Theta(m^2 n)$ which saves a factor of n comparing to the standard matrix-vector multiplications. It can also be viewed as an opportunity for vector-parallelism as illustrated below:

$$\mathrm{DFT}_2 \otimes I_4 = \begin{bmatrix} 1 & & & & 1 & & & \\ & 1 & & & & 1 & & \\ & & 1 & & & & 1 & \\ & & & 1 & & & & 1 \\ 1 & & & & -1 & & & \\ & 1 & & & & -1 & & \\ & & 1 & & & & -1 & \\ & & & 1 & & & & -1 \end{bmatrix}$$

in which the same operations are performed across the data vector (one addition and one subtraction here).

Definition 7. *The direct sum of A and B is an $(m+q) \times (n+s)$ matrix over* $\mathbf{k}$ *denoted by $A \oplus B$ and defined by*

$$A \oplus B = \begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix}. \tag{5.2}$$

More generally, for n matrices $A_0, \ldots, A_{n-1}$ over $\mathbf{k}$*, the direct sum of $A_0, \ldots, A_{n-1}$ is defined as*

$$\bigoplus_{i=0}^{n-1} A_i = A_0 \oplus A_1 \oplus \cdots \oplus A_{n-1} = \begin{bmatrix} A_0 & & & \\ & A_1 & & \\ & & \ddots & \\ & & & A_{n-1} \end{bmatrix}. \tag{5.3}$$

It is easy to show the following formula

$$I_n \otimes A = \oplus_{i=0}^{n-1} A = \mathrm{diag}(A, A, \ldots, A). \tag{5.4}$$

Definition 8. *The stride permutation $\mathcal{L}_m^{mn}$ permutes an input vector $\mathbf{x}$ of length mn as follows*

$$\mathbf{x}[in + j] \mapsto \mathbf{x}[jm + i], \tag{5.5}$$

for all $0 \leq i < m$ and $0 \leq j < n$. Let e_i be the vector of $\mathbf{k}^{mn}$ whose j-th entry is the Kronecker symbol δ_{ij} [1]. The matrix representation of $\mathcal{L}_m^{mn}$ in the basis $\{e_i \mid 1 \leq i \leq mn\}$ is denoted by L_m^{mn}. If $\mathbf{x}$ is viewed as an $n \times m$ matrix, then $\mathcal{L}_m^{mn}$ performs a transposition on this matrix.

Example 8. *Let $n = 4$ and $m = 2$. Then*

$$\mathcal{L}_2^8([x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7]) = [x_0, x_2, x_4, x_6, x_1, x_3, x_5, x_7]. \tag{5.6}$$

Thus $\{e_1, \ldots, e_8\}$ forms a basis of $V = \mathbf{k}^8$, and we have

$$\mathcal{L}_2^8(e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8) = (e_1, e_3, e_5, e_7, e_2, e_4, e_6, e_8). \tag{5.7}$$

The matrix representation of $\mathcal{L}_2^8$ in the basis $\{e_i \mid i = 1 \ldots 8\}$ is

$$L_2^8 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.8}$$

5.2.2 Discrete Fourier transform

In this subsection, we fix an integer $n \geq 2$. An element ω in $\mathbf{k}$ is an n-th primitive root of unity if n is the smallest positive integer such that $\omega^n = 1$ in $\mathbf{k}$.

Definition 9. *The n-point Discrete Fourier Transform (DFT) at ω is a linear map from the $\mathbf{k}$-vector space $\mathbf{k}^n$ to itself, defined by $\mathbf{x} \longmapsto \mathrm{DFT}_n \mathbf{x}$ with the n-th DFT*

[1] That is, $\delta_{ij} = 1$ if $i = j$ otherwise $\delta_{ij} = 0$.

matrix

$$\mathrm{DFT}_n = [\omega^{k\ell}]_{0\le k,\,\ell<n}. \tag{5.9}$$

In particular, the DFT of size 2 corresponds to the butterfly matrix

$$\mathrm{DFT}_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{5.10}$$

Given DFT_n, computing the product of DFT_n by $\mathbf{x}$ costs $\Theta(n^2)$ arithmetic operations in $\mathbf{k}$ via the standard linear algebra methods. The ability to compute DFTs fast is of great importance to a wide variety of applications, from digital signal processing and solving partial differential equations, etc.. The well-known Cooley-Tukey Fast Fourier Transform (FFT) [19] in its recursive form is a procedure for computing $\mathrm{DFT}_n\,\mathbf{x}$ based on the factorization of the matrix DFT_n.

Theorem 7. *For any integers positive q, s such that $n = qs$, we have:*

$$\mathrm{DFT}_{qs} = (\mathrm{DFT}_q \otimes I_s) D_{q,s} (I_q \otimes \mathrm{DFT}_s) L^{qs}_q, \tag{5.11}$$

where $D_{q,s}$ is the diagonal twiddle matrix defined as

$$D_{q,s} = \bigoplus_{j=0}^{q-1} \mathrm{diag}(1, \omega^j, \ldots, \omega^{j(s-1)}), \tag{5.12}$$

where ω is a n-th primitive root of unity.

Example 9. *Let $n = 4$ and ω be a 4-th primitive root of unity. The following matrix factorization illustrates Theorem 7.*

$$\begin{aligned}
\mathrm{DFT}_4 &= (\mathrm{DFT}_2 \otimes I_2) D_{2,2} (I_2 \otimes \mathrm{DFT}_2) L^4_2 \\
&= \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \omega \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & -1 & -\omega \\ 1 & -1 & 1 & -1 \\ 1 & -\omega & -1 & \omega \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix}.
\end{aligned}$$

Let $n = 2^k$ and let $C(n)$ be the number of arithmetic operations in $\mathbf{k}$ performed to compute $\mathrm{DFT}_n(\mathbf{x})$. We apply Equation 5.11 with $q = 2$ and $s = n/2$, obtaining

$$\mathrm{DFT}_n(\mathbf{x}) = (\mathrm{DFT}_2 \otimes I_{n/2}) D_{2,n/2} (I_2 \otimes \mathrm{DFT}_{n/2}) L^n_2(\mathbf{x}).$$

Observe that matrix L^n_2 only permutes the input data $\mathbf{x}$, $D_{2,n/2}$ scales a vector and $\mathrm{DFT}_2 \otimes I_{n/2}$ on a vector costs $\Theta(n)$ operations. Therefore, we have

$$C(n) = 2C(n/2) + \Theta(n),$$

which gives $C(n) = \Theta(n \log n)$.

5.2.3 FFTs over finite fields

As mentioned in the introduction, for FFT computations, the case where $\mathbf{k}$ is a finite field offers specific challenges in comparison to the case where $\mathbf{k}$ is the field $\mathbb{C}$ of complex numbers. This explains why FFT techniques used for numerical computation do not adapt straightforwardly to symbolic computation.

On the algebraic side, the most obvious difference is that the prime field $\mathbb{Z}/p\mathbb{Z}$ admits an n-th primitive root of unity if and only if n divides $p - 1$. In addition, radix 2 FFTs are often preferred to others in symbolic computation, since many algorithms, such as those for polynomial factorization, require the prime p to be small, say $p = 3, 5, 7, \ldots$. Since the radix must be invertible in $\mathbb{Z}/p\mathbb{Z}$, this essentially imposes the restriction to radix 2 FFTs. See [34] for details on these algebraic considerations.

On the implementation side, multiplying two elements a, b of $\mathbb{Z}/p\mathbb{Z}$ is obviously a key routine. Unlike the case of single and double precision floating point arithmetic, the operation $(a, b, p) \longmapsto (ab) \mod p$, for $a, b, p \in \mathbb{Z}$, is not provided directly by hardware. This operation is thus an efficiency-critical low-level software routine that the programmer has to take care of. When p is a machine word size prime, which is the assumption in this chapter, two techniques are popular in the symbolic computation community.

The first one takes advantage of hardware floating point arithmetic, see [27]. We call `double_mul_mod` our implementation of this technique, for which our C code is shown below. The fourth argument *pinv* is the inverse of p which is precomputed in floating point.

```
int double_mul_mod(int a, int b, int p, double pinv) {
   int q  = (int) ((((double) a) * ((double) b)) * pinv);
```

```
    int res = a * b - q * p;
    return (res < 0) ? (-res) : res;
}
```

In our implementation, double precision floating point numbers are encoded on 64 bits and make this technique work correctly for primes p up to 30 bits.

The second technique, called the Montgomery reduction [62], relies only on hardware integer arithmetic. We summarize this elegant trick. Consider a positive integer $R \geq p$ such that $\mathsf{gcd}(R, p) = 1$. Hence there exists integers R^{-1}, p' such that we have:

$$RR^{-1} - p\,p' = 1 \quad \text{and} \quad 0 < p' < R. \tag{5.13}$$

Consider an integer x, satisfying $0 \leq x < p^2$, and for which we want to compute $x/R \mod p$. Let c and d (resp. e and f) be the quotient and remainder in the Euclidean division of x by R (resp. dp' and R). Then, it is easy to prove that there exists an integer q such that $x + fp = qR$ holds, that is, satisfying $q \equiv x/R \mod p$. If $p > 2$ then R can be chosen to be a power of 2. Therefore, with this choice, computing $x/R \mod p$ amounts to 2 multiplications, 2 additions and 3 shifts. Now, in order to compute products in $\mathbb{Z}/p\mathbb{Z}$, one "represents" any residue class $a \mod p$ by $aR \mod p$. Then, applying the previous trick to $x = (aR)(bR) \mod p$ one obtains efficiently $(ab)/R \mod p$, that is, the representative of $(ab) \mod p$. An improved version of this trick was proposed in [53].

5.3 Iterative FFT formulas

There are abundant variations of the FFT algorithm which run in $\Theta(n \log n)$ time. However, the program structures and the performance vary greatly for different hardware architectures. The idea of choosing suitable FFT algorithm for different (serial or parallel) architectures has been applied thoroughly in the SPIRAL project [32, 72]. In this section we present two iterative FFT algorithms which will be implemented and examined in detail.

Assume that n is a power of 2, say $n = 2^k$. Applying Formula (5.11) once with $q = 2$ and $s = n/2$, we have

$$\mathrm{DFT}_n = (\mathrm{DFT}_2 \otimes I_{n/2}) D_{2,n/2} (I_2 \otimes \mathrm{DFT}_{n/2}) L^n_2. \tag{5.14}$$

Before continuing to unroll the DFT matrix $\mathrm{DFT}_{n/2}$, we introduce a new notation to

simplify the presentation. For integers $i, j, h \geq 1$, define

$$\Delta(i, j, h) = (I_i \otimes \mathrm{DFT}_j \otimes I_h) \tag{5.15}$$

which is a square matrix of size ijh. Applying the same factorization to $n/2 = 2 \times n/4$ and combining with Equation 5.14, we have

$$\begin{aligned}
\mathrm{DFT}_n &= (\mathrm{DFT}_2 \otimes I_{n/2}) D_{2,n/2} (I_2 \otimes ((\mathrm{DFT}_2 \otimes I_{n/4}) D_{2,n/4} (I_2 \otimes \mathrm{DFT}_{n/4}) L_2^{n/2})) L_2^n \\
&= (\mathrm{DFT}_2 \otimes I_{n/2}) D_{2,n/2} (I_2 \otimes \mathrm{DFT}_2 \otimes I_{n/4}) (I_2 \otimes D_{2,n/4}) (I_4 \otimes \mathrm{DFT}_{n/4}) (I_2 \otimes L_2^{n/2}) L_2^n \\
&= \underbrace{\Delta(1, 2, 2^{k-1}) D_{2,2^{k-1}}} \underbrace{\Delta(2, 2, 2^{k-2})(I_2 \otimes D_{2,2^{k-2}})} (I_4 \otimes \mathrm{DFT}_{n/4}) (I_2 \otimes L_2^{n/2}) L_2^n.
\end{aligned}$$

Continuing to unroll $I_4 \otimes \mathrm{DFT}_{n/4}$ with the factorization $n/4 = 2 \times n/8$, until reaching the base case DFT_m with some $m \mid n$), we have the following theorem.

Theorem 8 (Iterative Cooley-Tukey DFT factorization with a base case)**.** *Let $n = 2^k$ and $m = 2^\ell$ such that $0 < \ell < k$. The following matrix factorization holds:* [2]

$$\mathrm{DFT}_{2^k} = \left(\prod_{i=0}^{k-\ell-1} \Delta\left(2^i, 2, 2^{k-i-1}\right) \left(I_{2^i} \otimes D_{2,2^{k-i-1}}\right) \right) (I_{2^{k-\ell}} \otimes \mathrm{DFT}_m) \prod_{i=k-\ell-1}^{0} I_{2^i} \otimes L_2^{2^{k-i}}. \tag{5.16}$$

When the base size $m = 2$, then the above factorization reduces to the well-known iterative Cooley-Tukey factorization

$$\mathrm{DFT}_{2^k} = \left(\prod_{i=0}^{k-1} \Delta\left(2^i, 2, 2^{k-i-1}\right) \left(I_{2^i} \otimes D_{2,2^{k-i-1}}\right) \right) \prod_{i=k-1}^{0} (I_{2^i} \otimes L_2^{2^{k-i}}). \tag{5.17}$$

Given Equation 5.16 or Equation 5.17, computing $\mathrm{DFT}_n(\mathbf{x})$ is not through matrix-vector multiplications, since $\Delta(2^i, 2, 2^{k-i-1}) = I_{2^i} \otimes \mathrm{DFT}_2 \otimes I_{2^{k-i-1}}$, $I_{2^i} \otimes D_{2,2^{k-i-1}}$ and $I_{2^i} \otimes L_2^{2^{k-i}}$ are all $n \times n$ structured sparse matrices. The cost of applying these matrices on $\mathbf{x}$ are all linear in n. Expressing a FFT algorithm in terms of DFT matrix factorizations also provides solid understanding towards the algorithm. In the following paragraphs, we present in details how those matrices operate on the input data $\mathbf{x}$. In the later sections, we report our implementation details.

Permutation By definition, how $I_{2^i} \otimes L_2^{2^{k-i}}$ operates on $\mathbf{x}$ is equivalent to treating $\mathbf{x}$ as 2^i equal-length subvectors of size 2^{k-i} and applying $L_2^{2^{k-i}}$ on each subvector.

[2] Matrix multiplications are not commutative. Throughout, the product $\prod_{i=1}^{s} M_i$ stands for $M_1 M_2 \cdots M_s$, while $\prod_{i=s}^{1} M_i$ means $M_s M_{s-1} \cdots M_1$.

According to Definition 8, $L_2^{2^{k-i}}$ permutes a subvector of $\mathbf{x}$, or transposes a subvector of $\mathbf{x}$ when treating it as a $2^{k-i-1} \times 2$ matrix.

Example 10. *Let $k = 4$, $i = 2$ and $\mathbf{x} = (0, 1, 2, \ldots, 15)$ be a vector of length $n = 16$. Then $I_{2^i} \otimes L_2^{2^{k-i}} = I_4 \otimes L_2^4$ permute $\mathbf{x}$ as follows:*

$$\begin{aligned} (0,1,2,3) &\Longrightarrow (0,2,1,3) \\ (4,5,6,7) &\Longrightarrow (4,6,5,7) \\ (8,9,10,11) &\Longrightarrow (8,10,9,11) \\ (12,13,14,15) &\Longrightarrow (12,14,13,15), \end{aligned}$$

from which it follows $I_4 \otimes L_2^4(\mathbf{x}) = (0, 2, 1, 3, 4, 6, 5, 7, 8, 10, 9, 11, 12, 14, 13, 15)$.

Twiddling By Definition 5.12, the matrix $D_{2,2^{k-i-1}}$ is an $2s \times 2s$ diagonal matrix of the form

$$D = \begin{bmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & 1 & & & & \\ & & & & \omega^i & & & \\ & & & & & \ddots & & \\ & & & & & & \omega^{i(s-1)} \end{bmatrix},$$

where $s = 2^{k-i-1}$ and ω is a n-th primitive root of unity. Thus $I_{2^i} \otimes D_{2,2^{k-i-1}}$ is an $n \times n$ diagonal matrix with 2^i copies of D along the diagonal.

Butterfly By the definition of tensor product, $\mathrm{DFT}_2 \otimes I_{2^{k-i-1}}$ is a $2s \times 2s$ square matrix with $s = 2^{k-i-1}$ of the form

$$B = \begin{bmatrix} 1 & & & 1 & & & \\ & \ddots & & & \ddots & & \\ & & 1 & & & 1 & \\ 1 & & & -1 & & & \\ & \ddots & & & \ddots & & \\ & & 1 & & & -1 \end{bmatrix},$$

Thus $I_{2^i} \otimes \mathrm{DFT}_2 \otimes I_{2^{k-i-1}}$ is an $n \times n$ matrix with 2^i copies of B along the diagonal.

Remark 5. *Formula (5.16) reduces the computation of DFT_{2^k} to composing lists*

of permutations, DFT_m*, diagonal twiddling and butterflies. This is the basis of the implementation presented in Section 5.4. which needs the assumption that the basis size* m *is a multiple of* 16.

There exists another way to factorize n, exactly opposite to the process of deriving Formula (5.16). That is, we apply Theorem 7 with $n = n/2 \times 2$, $n/2 = n/4 \times 2$, etc:

$$\begin{aligned}
\mathrm{DFT}_{2^k} &= (\mathrm{DFT}_2 \otimes I_{2^{k-1}})(D_{2,2^{k-1}} \otimes I_1)(L_2^{2^k} \otimes I_1) \\
&\quad (\mathrm{DFT}_{2^{k-1}} \otimes I_2) \\
&= (\mathrm{DFT}_2 \otimes I_{2^{k-1}})(D_{2,2^{k-1}} \otimes I_1)(L_2^{2^k} \otimes I_1) \\
&\quad (\mathrm{DFT}_2 \otimes I_{2^{k-1}})(D_{2,2^{k-2}} \otimes I_2)(L_2^{2^{k-1}} \otimes I_2) \\
&\quad (\mathrm{DFT}_{2^{k-2}} \otimes I_4) \\
&= (\mathrm{DFT}_2 \otimes I_{2^{k-1}})(D_{2,2^{k-1}} \otimes I_1)(L_2^{2^k} \otimes I_1) \\
&\quad (\mathrm{DFT}_2 \otimes I_{2^{k-1}})(D_{2,2^{k-2}} \otimes I_2)(L_2^{2^{k-1}} \otimes I_2) \\
&\quad (\mathrm{DFT}_2 \otimes I_{2^{k-1}})(D_{2,2^{k-3}} \otimes I_4)(L_2^{2^{k-2}} \otimes I_4) \\
&\quad (\mathrm{DFT}_{2^{k-3}} \otimes I_8) \\
&= \cdots \\
&= \prod_{i=0}^{k-1} (\mathrm{DFT}_2 \otimes I_{2^{k-1}})(D_{2,2^{k-i-1}} \otimes I_{2^i})(L_2^{2^{k-i}} \otimes I_{2^i}),
\end{aligned}$$

from which we derive another factorization of DFT_n.

Theorem 9 (Iterative Stockham DFT factorization)**.** *The matrix* DFT_{2^k} *can be written as a product of matrices*

$$\mathrm{DFT}_{2^k} = \prod_{i=0}^{k-1} (\mathrm{DFT}_2 \otimes I_{2^{k-1}})(D_{2,2^{k-i-1}} \otimes I_{2^i})(L_2^{2^{k-i}} \otimes I_{2^i}). \tag{5.18}$$

The Stockham FFT factorization can be found in the paper [78]. Comparing with the iterative Cooley-Tukey factorization, there is a key difference. For the Cooley-Tukey factorization, the identity matrix I_s only appears on the left for twiddling and permutation. For the Stockham factorization, the identity matrix I_s only appears on the right. As we shall see in the later sections, this difference brings a significant performance gap. The Stockham factorization based implementation will be presented in Section 5.5.

5.4 Implementation of the Cooley-Tukey FFT

We stick to the notations and hypotheses introduced in Section 5.2.2. Our purpose is to describe our CUDA implementation of Formula (5.16). The idea behind this formula is that the base case DFT_m can be implemented efficiently, for m small enough, typically $m = 16$. [3] This formula can be interpreted as the composition of three computational steps:

$\mathbf{S}_1$: $\mathbf{x} \longmapsto \prod_{i=k-\ell-1}^{0} (I_{2^i} \otimes L_2^{2^{k-i}})\mathbf{x}$,

$\mathbf{S}_2$: $\mathbf{x} \longmapsto (I_{2^{k-\ell}} \otimes \mathrm{DFT}_m)\mathbf{x}$,

$\mathbf{S}_3$: $\mathbf{x} \longmapsto \prod_{i=0}^{k-\ell-1} \left(I_{2^i} \otimes \mathrm{DFT}_2 \otimes I_{2^{k-i-1}}\right) \left(I_{2^i} \otimes D_{2,2^{k-i-1}}\right) \mathbf{x}$.

According to the definition, the step $\mathbf{S}_2$ essentially reduces to execute a sequence of base DFT_m, each of which operates on a sub-vector of $\mathbf{x}$, independently. Therefore, we focus hereafter on $\mathbf{S}_1$ and $\mathbf{S}_3$. Note that in step $\mathbf{S}_3$, we fuse the twiddling with the butterfly to reduce memory accesses. For steps $\mathbf{S}_1$ and $\mathbf{S}_2$ we need to double-buffer the array to avoid synchronizations among different CUDA thread blocks, see Section B.1 for related discussions. That is, at the same time, we have two vectors X and Y of length n, one of which is the input and the other is the output, and they switch their role after a kernel application on the input.

Implementation of step $\mathbf{S}_1$

Step $\mathbf{S}_1$ consists of a sequence of calls to the following GPU kernel, with s ranging from 1 to $\frac{n}{2m}$. Its specification is

```
/**
 * Compute Y = (I_s x L_2^{n/s})X
 *
 * @X, input array of length n
 * @Y, output array of length n
 */
void list_transpose_kernel(int *Y, int *X, int n, int s);
```

Applying the matrix $I_s \otimes L_2^{n/s}$ on a vector $\mathbf{x}$ of length n is equivalent to

[3] There are several reasons to set m being a multiple of 16, mainly from the implementation point of view. Firstly, in CUDA, a basic unit that could be scheduled is called a warp, 16 threads with consecutive thread indices. For $m = 2^\ell \geq 16$, memory accesses to the GPU global memory are easier to get well-aligned. Secondly, the kernel for permuting data has been simplified due to the choice of m, where no complicated internal permutations are needed.

(i) dividing $\mathbf{x}$ evenly into s sub-vectors,

(ii) regarding each sub-vector as a $\frac{n}{2s} \times 2$ matrix and transposing it.

Therefore, step $\mathbf{S}_1$ essentially consists of s matrix transpositions of size $\frac{n}{2s} \times 2$. Following the spirit of [75] for matrix transposition, we realized an efficient subroutine to transpose a list of matrices. Note that we could not directly adapt their code since each matrix has only two columns. Without padding the input data with zeros, our implementation is still able to utilize the shared memory of CUDA devices effectively. For simplicity, we present our implementation with the following example.

Example 11. *Let M be a 16×2 matrix. We set the thread block size to 16×2 with indices (i, j) for $0 \leq i < 16$ and $j = 0, 1$. Then we first read M into an array M_s of size 32 residing in the shared memory space as follows*

```
int i = threadIdx.y * 16 + threadIdx.x;
M_s[i] = M[i];
```

That is, the above segment of code transforms M into the shared array M_s via two coalesced reads, without changing the data layout. Still, we look at the shared array M_s as a 16×2 matrix, then we achieve the transposition by writing the data back to the global memory column-wise as follows

```
int i = threadIdx.y * 16 + threadIdx.x;
M[i] = M_s[threadIdx.x * 2 + threadIdx.y];
```

The first 16 threads (a half warp) $\{(i, 0) \mid 0 \leq i < 16\}$ read in $M_s[0], M_s[2], \ldots, M_s[30]$, and write to $M[0], M[1], \ldots, M[15]$. On the other hand, the second half warp of threads $\{(i, 1) \mid 0 \leq i < 16\}$ read in $M_s[1], M_s[3], \ldots, M_s[31]$ and write to $M[16], M[17], \ldots, M[31]$. Again all the writes to global memory are coalesced.

The above example can be generalized to transpose a list of $m \times 2$ matrices with only coalesced reads and writes for any $m \geq 16$, which satisfies the specification of the kernel `list_transpose_kernel`.

5.4.1 Implementation of step $\mathbf{S}_3$

We are going to map the formula

$$(I_{2^i} \otimes \mathrm{DFT}_2 \otimes I_{2^{k-i-1}})(I_{2^i} \otimes D_{2,2^{k-i-1}}), \quad 0 \leq i \leq k - \ell - 1 \tag{5.19}$$

to a GPU kernel. According to Proposition 8, the following relation holds

$$(I_{2^i} \otimes \mathrm{DFT}_2 \otimes I_{2^{k-i-1}})(I_{2^i} \otimes D_{2,2^{k-i-1}}) = I_{2^i} \otimes \left((\mathrm{DFT}_2 \otimes I_{2^{k-i-1}})D_{2,2^{k-i-1}}\right) \quad (5.20)$$

Hence step $\mathbf{S}_3$ consists of a sequence of calls to the following GPU kernel, with q ranging from $\frac{n}{2}$ to m. Its specification is

```
/**
 * Compute Y = (I_{n/2q} x (DFT_2 x I_q) D_{2, q}) X
 *
 * @X, input array of length n
 * @Y, output array of length n
 */
void list_butterfly_kernel(int *Y, int *X, int n, int q);
```

We notice that $(\mathrm{DFT}_2 \otimes I_q)D_{2,q}$ is, in fact, the *classical butterfly* operation, which can be realized as,

```
for (i = 0; i < q; ++i) {
    Y[i]   = X[i] + X[q+i] * W[i];
    Y[q+i] = X[i] - X[q+i] * W[i];
}
```

with $W[i] = w^i$ and w is a $(2q)$-th primitive root of unity. The formula $(\mathrm{DFT}_2 \otimes I_q)D_{2,q}$ will be applied to a segment of data of length $2q$. Hence, with $n/2$ threads, one can realize `list_butterfly_kernel` which implements the formula $I_{2^{i-1}} \otimes \left((\mathrm{DFT}_2 \otimes I_{2^{k-i}})D_{2,2^{k-i}}\right)$ for each i. The following simplified kernel handles the case where one thread block performs more than one groups of butterflies.

```
__global__ void
list_butterfly_kernel_a(int q, int *Y, int *X, int w, int p)
{
    int gval = threadIdx.x / q;         // group index
    int rval = threadIdx.x % q;         // index inside a group

    int offset = blockIdx.x * 2 * blockDim.x + gval * 2 * q + rval;
    int x1 = X[offset];
    int x2 = X[offset + q];

    int wi = pow_mod(w, rval, p);       // twiddle factor
```

```
    int t = mul_mod(x2, wi, p);        // t = x2 * wi mod p
    Y[offset] = add_mod(x1, t, p);     // y1 = x1 + t mod p
    Y[offset + q] = sub_mod(x1, t, p); // y2 = x1 - t mod p
}
```

5.5 Implementation of the Stockham FFT

Recall that the Stockham FFT factorization (5.18) is

$$\mathrm{DFT}_{2^k} = \prod_{i=0}^{k-1} (\mathrm{DFT}_2 \otimes I_{2^{k-1}})(D_{2,2^{k-i-1}} \otimes I_{2^i})(L_2^{2^{k-i}} \otimes I_{2^i}).$$

For each fixed $0 \leq i < k$, there are three computational steps:

$\mathbf{A}_1$**:** $\mathbf{x} \longmapsto (L_2^{2^{k-i}} \otimes I_{2^i})\mathbf{x}$,

$\mathbf{A}_2$**:** $\mathbf{x} \longmapsto (D_{2,2^{k-i-1}} \otimes I_{2^i})\mathbf{x}$,

$\mathbf{A}_3$**:** $\mathbf{x} \longmapsto (\mathrm{DFT}_2 \otimes I_{2^{k-1}})\mathbf{x}$.

Comparing to the Cooley-Tukey FFT, the Steps $\mathbf{A}_2$ and $\mathbf{A}_3$ are relatively easy to be mapped into GPU kernels, while it is rather tricky to map Step $\mathbf{A}_1$ into GPU kernels. Similar to the implementation of the Cooley-Tukey FFT, we use double-buffers in these steps.

5.5.1 Implementation of step $\mathbf{A}_1$

We describe how to map the formula $L_2^{n/s} \otimes I_s$ to GPU kernels, where n is the FFT size and s is called the stride size. Let M be an $(n/s - 1) \times 2s$ matrix stored in the row-major layout. The effect of this stride permutation on M is to perform the following reordering:

$$M = \begin{bmatrix} \mathbf{S}_0 & \mathbf{S}_1 \\ \mathbf{S}_2 & \mathbf{S}_3 \\ \mathbf{S}_4 & \mathbf{S}_5 \\ \vdots & \vdots \\ \mathbf{S}_{(n/s-2)} & \mathbf{S}_{(n/s-1)} \end{bmatrix} \Longrightarrow T = \begin{bmatrix} \mathbf{S}_0 & \mathbf{S}_2 & \mathbf{S}_4 & \cdots & \mathbf{S}_{(n/s-2)} \\ \mathbf{S}_1 & \mathbf{S}_3 & \mathbf{S}_5 & \cdots & \mathbf{S}_{(n/s-1)} \end{bmatrix}$$

where $\mathbf{S}_i$, called a slice, denotes the elements of M with indices $is \cdots (is + s - 1)$. When we regard M as an $(n/s - 1) \times 2$ matrix (each $\mathbf{S}_i$ is a single element of M), the output matrix T, of size $2 \times (n/s - 1)$, is the matrix transposition of M.

Example 12. *Consider $n = 2^6 = 64$. Then all the formulas are $L_2^{64} \otimes I_1$, $L_2^{32} \otimes I_2$, $L_2^{16} \otimes I_4$, $L_2^8 \otimes I_8$, $L_2^4 \otimes I_{16}$, and $L_2^2 \otimes I_{32}$. Some of these transpositions are:*

$$L_2^{32} \otimes I_2 : \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ \vdots & \vdots & \vdots & \vdots \\ 60 & 61 & 62 & 63 \end{bmatrix} \Longrightarrow \begin{bmatrix} 0 & 1 & 4 & 5 & \cdots & 60 & 61 \\ 2 & 3 & 6 & 7 & \cdots & 62 & 63 \end{bmatrix}$$

$$L_2^8 \otimes I_8 : \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & \cdots & 15 \\ 16 & 17 & 18 & 19 & 20 & \cdots & 31 \\ 32 & 33 & 34 & 35 & 36 & \cdots & 47 \\ 48 & 49 & 50 & 51 & 52 & \cdots & 63 \end{bmatrix} \Longrightarrow \begin{bmatrix} 0 & \cdots & 7 & 16 & \cdots & 23 & 32 & \cdots & 39 & 48 & \cdots & 55 \\ 8 & \cdots & 15 & 24 & \cdots & 31 & 40 & \cdots & 47 & 56 & \cdots & 63 \end{bmatrix}$$

$$L_2^4 \otimes I_{16} : \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & \cdots & 31 \\ 32 & 33 & 34 & 35 & 36 & \cdots & 63 \end{bmatrix} \Longrightarrow \begin{bmatrix} 0 & 1 & 2 & 3 & \cdots & 15 & 32 & \cdots & 47 \\ 16 & 17 & 18 & 19 & \cdots & 31 & 48 & \cdots & 63 \end{bmatrix}$$

As shown above, for a thread block of some fixed number of threads, the map between threads and element of the matrix are rather complex.

To achieve coalesced memory accesses for all threads, we use the shared memory space. Let τ be the number of threads in a thread block, typically $\tau = 128$. To use the shared memory space efficiently, τ should be a multiple of 16. Under the above setting, the number of threads blocks required is given by $\lambda = \frac{n}{\tau}$. We need to distinguish the following two cases

(1) $s \geq \tau$, that is, $\delta = \frac{s}{\tau}$ blocks are needed to move a slice of length s,

(2) $s < \tau$, that is, a thread block moves $\delta = \frac{\tau}{s}$ slices of data.

The reason to have such a case discussion is that the relation between τ and s determines the behavior of each thread block, specified in detail as follows.

Proposition 9 (Case $s \geq \tau$)**.** *Given a thread block of index $0 \leq i < \lambda$, we define*

$$i_q = \mathrm{quo}(i, \delta) \quad \textit{and} \quad i_r = \mathrm{rem}(i, \delta).$$

Then the output offset for the thread block i is given by the following formula:

$$\mathrm{rem}(i_q, 2) * \frac{n}{2} + \mathrm{quo}(i_q, 2) * s + i_r * \tau. \tag{5.21}$$

Moreover, each thread block does a direct copy.

Proof. Here i_q determines the slice index which thread block i is working on and i_r determines the offset inside this slice. If i_q is a multiple of 2 then $\mathbf{S}_{i_q}$ appears in the first row of the output matrix, otherwise it appears in the second row. Since $s \geq \tau$, each thread block only move a portion of a slice. Hence threads in a block move data directly, without performing any in-block shuffle. □

Example 13. *Again let $n = 64$, $s = 16$ and $\mathbf{x} = (0, 1, \ldots, 63)$. Assume the number of threads in a block is $\tau = 8$. Then the number of blocks is $\frac{n}{\tau} = 8$, two of which move one slice. Then $L^4_2 \otimes I_{16}$ permutes $\mathbf{x}$ as follows*

$$\begin{bmatrix} \mathbf{S}_0 & \mathbf{S}_1 \\ \mathbf{S}_2 & \mathbf{S}_3 \end{bmatrix} \Longrightarrow \begin{bmatrix} \mathbf{S}_0 & \mathbf{S}_2 \\ \mathbf{S}_1 & \mathbf{S}_3 \end{bmatrix}$$

where $\mathbf{S}_i = (16i, \ldots, 16i + 15)$ for $i = 0$, 1, 2 and 3. Thus thread blocks 0 and 1 move $\mathbf{S}_0$, blocks 2 and 3 moves $\mathbf{S}_1$, blocks 4 and 5 move $\mathbf{S}_2$, and blocks 6 and 7 move $\mathbf{S}_3$. Threads in each block move data according to the offset for the block and its thread index.

Proposition 10 (Case $s < \tau$)**.** *Given a thread block of index $0 \leq i < \lambda$, there are two output offsets for threads in the block*

$$\mathit{offset}_0 = i * \mathrm{quo}(\tau, 2) \quad \textit{and} \quad \mathit{offset}_1 = i * \mathrm{quo}(\tau, 2) + \frac{n}{2}. \tag{5.22}$$

Those $\mathbf{S}_i$'s with even indices will be moved with offset_0, while the others will be moved with offset_1.

Proof. For each thread block, half of the threads moves data to the first row and the remaining half moves data to the second row. Then the conclusion follows from the fact that each row consists of $n/2$ elements. □

Example 14. *Let* $n = 64$, $s = 4$ *and* $\mathbf{x} = (0, 1, \ldots, 63)$. *Assume the number of threads in a block is* $\tau = 8$. *Then the number of blocks is* $\frac{n}{\tau} = 8$, *each of which moves* 2 *slices. Then* $L^{16}_2 \otimes I_4$ *permutes* $\mathbf{x}$ *as follows*

$$\begin{bmatrix} \mathbf{S}_0 & \mathbf{S}_1 \\ \mathbf{S}_2 & \mathbf{S}_3 \\ \mathbf{S}_4 & \mathbf{S}_5 \\ \mathbf{S}_6 & \mathbf{S}_7 \\ \mathbf{S}_8 & \mathbf{S}_9 \\ \mathbf{S}_{10} & \mathbf{S}_{11} \\ \mathbf{S}_{12} & \mathbf{S}_{13} \\ \mathbf{S}_{14} & \mathbf{S}_{15} \end{bmatrix} \Longrightarrow \begin{bmatrix} \mathbf{S}_0 & \mathbf{S}_2 & \mathbf{S}_4 & \mathbf{S}_6 & \mathbf{S}_8 & \mathbf{S}_{10} & \mathbf{S}_{12} & \mathbf{S}_{14} \\ \mathbf{S}_1 & \mathbf{S}_3 & \mathbf{S}_5 & \mathbf{S}_7 & \mathbf{S}_9 & \mathbf{S}_{11} & \mathbf{S}_{13} & \mathbf{S}_{15} \end{bmatrix}$$

where $\mathbf{S}_i = (4i, \ldots, 4i + 3)$ *for* $i = 0 \cdots 15$. *Thread block* i *moves slices* $\mathbf{S}_{2i}$ *and* $\mathbf{S}_{2i+1}$, *where* $\mathbf{S}_{2i}$ *uses* offset_0 *and* $\mathbf{S}_{2i+1}$ *uses* offset_1.

We note that in the implementation each thread block first reads $\frac{\tau}{s}$ consecutive slices into the shared memory, then performs an in-block shuffle inside the shared memory, and in the end writes slices back with coalesced writes. This approach achieves high performance in terms of memory accesses.

5.5.2 Implementation of steps $\mathbf{A}_2$ and $\mathbf{A}_3$

According to its definition, $D_{2,2^{k-i-1}}$ is a diagonal matrix of size 2^{k-i} and thus $D_{2,2^{k-i-1}} \otimes I_{2^i}$ is again a diagonal matrix of size n, with each diagonal element repeated 2^i times. Hence step $\mathbf{A}_2$ simply scales $\mathbf{x}$ with powers of the primitive root of unity ω. On the other hand, step $\mathbf{A}_3$ is a list of basic butterflies with stride size $n/2$. This step accesses data in a very uniform manner. In the following section, we discuss the performance implications of steps $\mathbf{A}_2$ and $\mathbf{A}_3$.

5.6 Experimentation

We have realized in CUDA 2.2 both Cooley-Tukey FFT and Stockham FFT, and conducted a series of benchmarks using a Geforce GTX 285 graphics card on a desktop with the processor Intel Core 2 Quad CPU Q9400 @ 2.66GHz and 6 GB main memory. This graphics card has the compute capability 1.3, consists of 30 multiprocessors, each of which has 8 cores for integer and single-precision floating-point arithmetic operations. However, each multiprocessor has only 1 double-precision floating-point unit.

This is an important characteristic for our `double_mul_mod` routine implementing the map $(a, b, p) \mapsto (ab) \mod p$, as described Section 5.2.3. We note that for the most recent Nvidia graphics cards having the compute capability 2.x, the ability to perform double-precision floating-point operations has been greatly enhanced.

The experimentation is described in the following three subsections. Section 5.6.1 is dedicated to modular multiplication, and more precisely, to a comparative implementation of the map $(a, b, p) \mapsto (ab) \mod p$ on both CPU and GPU. Section 5.6.2 presents the results for our GPU implementation of the FFT formulas of Cooley-Tukey and Stockham, as described in Sections 5.4 and 5.5. Section 5.6.3 compares the performance of FFT-based univariate polynomial multiplication codes for CPU and GPU.

5.6.1 Modular multiplication

Figure 5.1 and Figure 5.2 are experimental results for modular multiplication on CPU and GPU respectively. In both cases, each slot of an input array of length $n = 2^k$ consisting of machine word size integers is multiplied by a given machine word size integer ω. This type of calculation is typical for FFT algorithms. For both CPU and GPU we compare our implementations of the Montgomery reduction and `double_mul_mod`. For the GPU kernel, we could choose to have a single thread or multiple threads. In our experimentation, we use the latter and in this case each multiprocessor can only process a double-precision floating-point operation at a time which downgrades the performance. Both the array length and the time are scaled by the base 2 logarithm.

Figure 5.1 shows that `double_mul_mod` is about 1.5 faster than the method based on the Montgomery reduction, when running serial C code on the CPU. On the GPU, Figure 5.2 shows that `double_mul_mod` is still slightly better than the method based on the Montgomery reduction, which is a surprise to us. When we increase the number of modular multiplications, the one relying on double-precision floating point computations outperforms the one relying on the Montgomery reduction. We do the same test as in Figure 5.1 on GPU, and run the kernels with massive threads.

5.6.2 Cooley-Tukey verses Stockham FFT

It is challenging to figure out what are the best implementation techniques for each of the two formulas. During our experiments, we realized that the pre-computation of the powers $1, \omega, \omega^2, \ldots, \omega^{n/2-1}$ is a necessity, for an n-point FFT. This step is rather

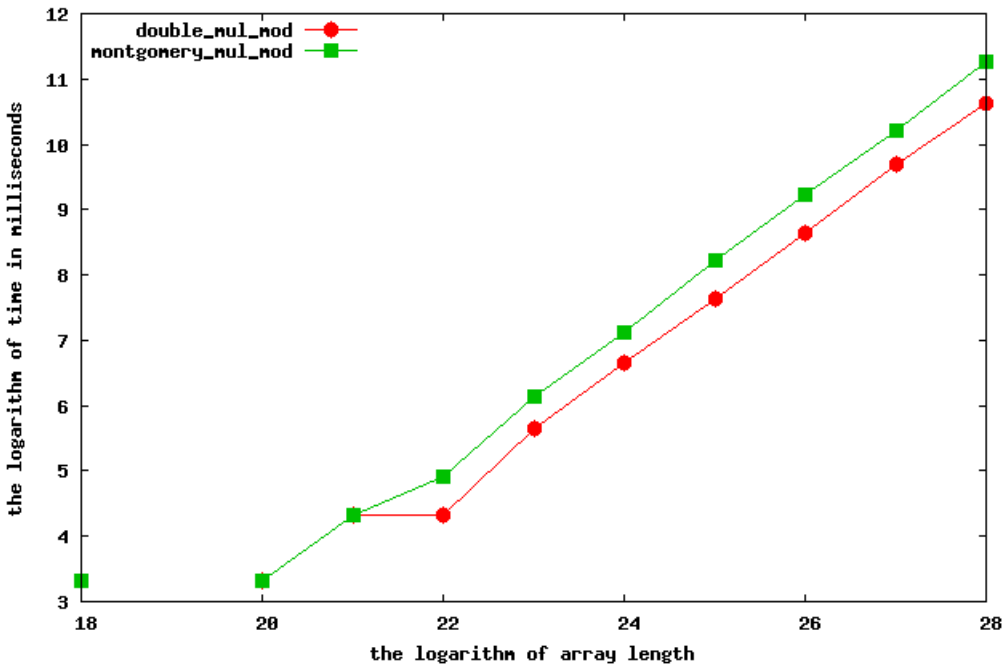

Figure 5.1: Modular multiplications on CPU

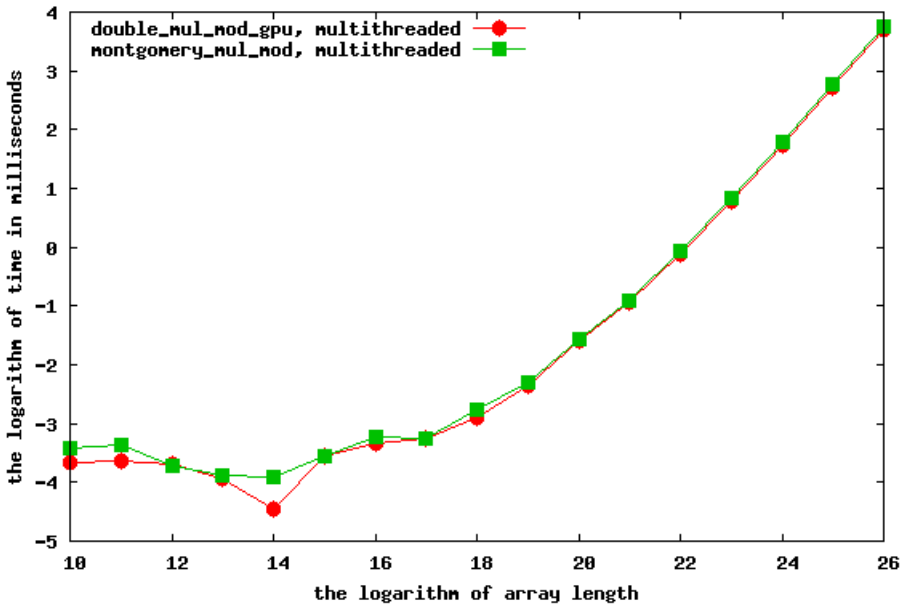

Figure 5.2: Modular multiplications on GPU

time-consuming if we implement it naively, that is, first compute those powers in the host sequentially, and then transfer them to the GPU device. Fortunately, this preprocessing is a special form of the exclusive prefix sum and the pre-computation can be achieved by a sequence of GPU kernel calls to a subroutine `double_expand`, which takes an array $\{1, \omega, \ldots, \omega^{s-1}\}$ of length s as input and returns $\{\omega^s, \ldots, \omega^{2s-1}\}$ by multiplying each element of the input array with ω^s.

We have noticed that step $\mathbf{S}_3$ of the Cooley-Tukey FFT (as described in Section 5.4.1) was accessing the powers of ω by performing larger and larger jumps. For example, while the following formula $(\mathrm{DFT}_2 \otimes I_{2^{k-i}})(I_{2^{i-1}} \otimes D_{2,2^{k-i}})$ operates on a sub-vector of length 2^{k-i+1}, the powers $\{1, \omega^{2^i}, (\omega^{2^i})^2, \ldots, (\omega^{2^i})^{2^{k-i}-1}\}$ get accessed. We call *jumped powers at level* i these latter powers. Therefore, we considered pre-computing not only the powers $\{1, \omega, \ldots, \omega^{n/2-1}\}$ but also all jumped powers at level i for each i.

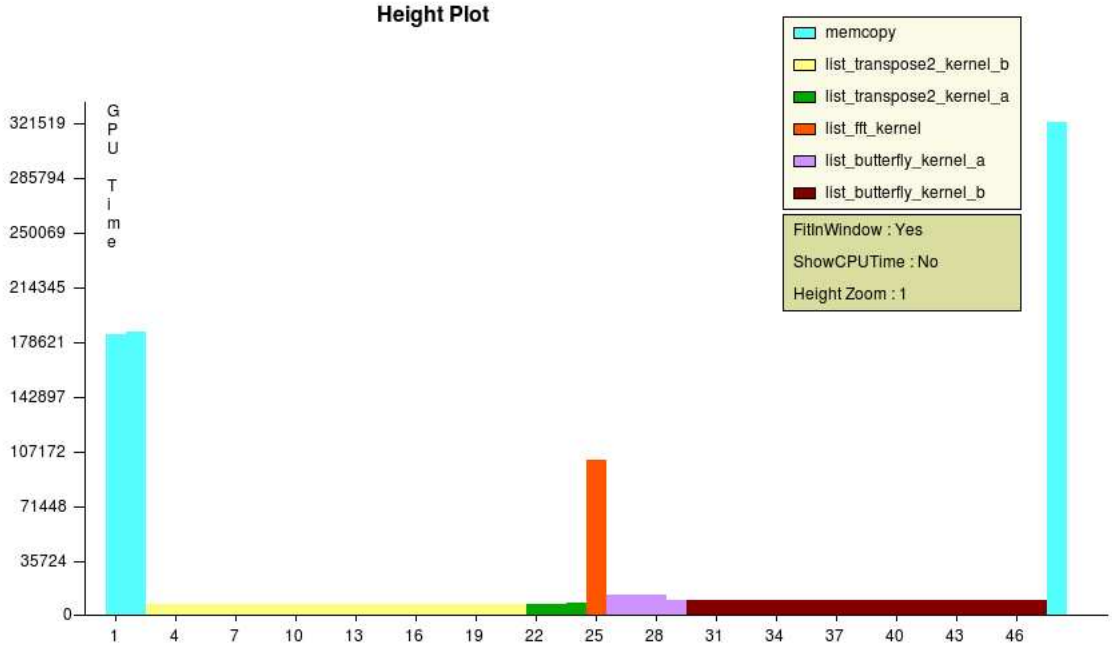

Figure 5.3: Cooley-Tukey FFT with pre-computed jumped powers

To visualize the performance of our implementation, we use the Nvidia's visual profiler `cudaprof` to analyze CUDA kernel calls. It is very helpful to find out the bottlenecks of an implementation. For instance, Figure 5.3 shows the kernel statistics where the pre-computation of jumped powers has been done on the host, In this figure, the x-axis shows the CUDA kernel call indices in chronological order and the y-axis is proportional to the GPU time for each kernel. The second kernel moves the

extra $n/2$ powers of the primitive root of unity to the device memory, which affects the overall performance. Note that those method names _a or _b as a suffix, since we implement the same algorithm for handling input data in different ranges.

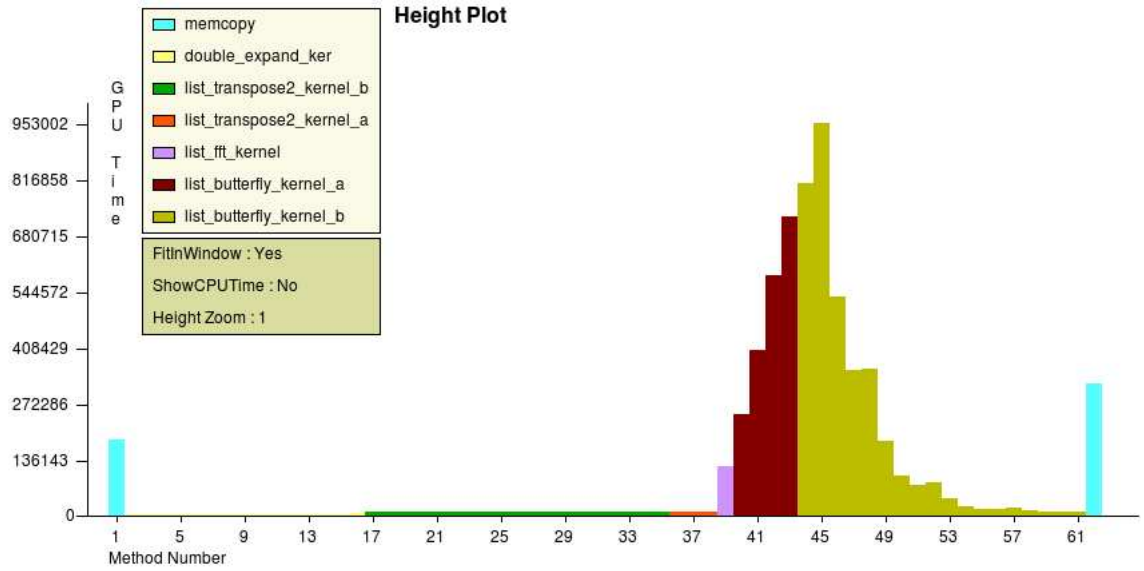

Figure 5.4: Cooley-Tukey FFT without pre-computed jumped powers

If the jumped powers were not computed in advance, the accesses to those powers harm the performance heavily as shown by Figure 5.4, since those memory accesses to the global memory are non-coalesced, in the step $\mathbf{S}_3$ of the Cooley-Tukey FFT. The time spent by the kernel `list_butterfly_kernel` increases significantly, which greatly downgrades the overall performance. To our knowledge, it is hard to achieve coalesced accesses without pre-computing jumped powers in implementing the Cooley-Tukey FFT.

However, the Stockham FFT avoids such a problem. Indeed, all the accesses to a power of ω are packed together, resulting in a broadcasting inside a thread block. Figure 5.5 shows the kernel statistics of the Stockham FFT of size 2^{26}, which is our best GPU univariate FFT implementation.

Without computing jumped powers, our Stockham FFT only pre-computes all powers use an extremely fast kernel `double_expand_ker`. The steps $\mathbf{A}_1$, $\mathbf{A}_2$ and $\mathbf{A}_3$ are realized by `stride_transpose2_kernel`, `stride_twiddle_kernel` and `butterfly_kernel`, respectively. The first and last kernels are for the input and output data transfer. All of them are running very efficiently.

For completeness, in Figure 5.6 we compare our two GPU implementations for FFT against our C code from `modpn` [50]. This latter library is shipped with the

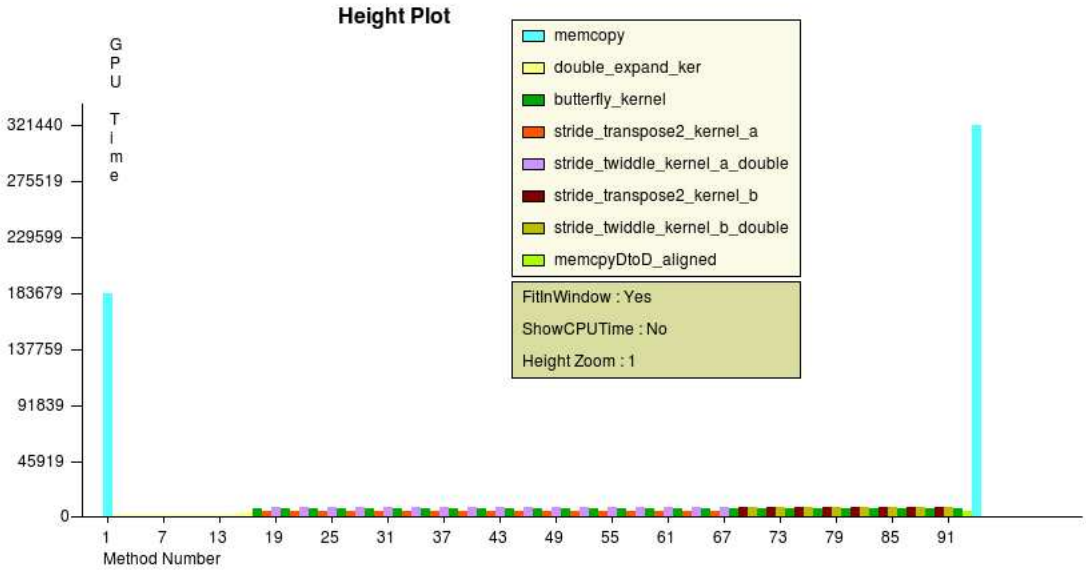

Figure 5.5: Stockham FFT on GPU

	CT	CT + transfer	Stockham	Stockham + transfer	`modpn`
12	1	1	2	2	1
13	2	2	2	3	1
14	1	2	2	3	3
15	2	2	3	3	4
16	3	3	3	4	10
17	4	5	3	5	16
18	6	9	4	7	37
19	11	15	6	10	71
20	22	28	9	16	174
21	44	56	16	28	470
22	83	105	29	52	997
23	165	210	56	101	2070
24	330	418	113	201	4194
25	667	842	230	405	8611
26	1338	1686	473	822	17617

Figure 5.6: Timing of FFT codes on CPU and GPU in milliseconds

computer algebra system MAPLE and is considered as a reference code for FFT computations over finite fields. The first column is the logarithm of FFT size in base 2. The second and the fourth columns show the timing of our Cooley-Tukey and Stockham FFT implementations, without counting the data transfer between GPU and CPU, respectively. The third and the fifth columns show these FFT implementations with the data transfer. The last column shows the `modpn` FFT timing.

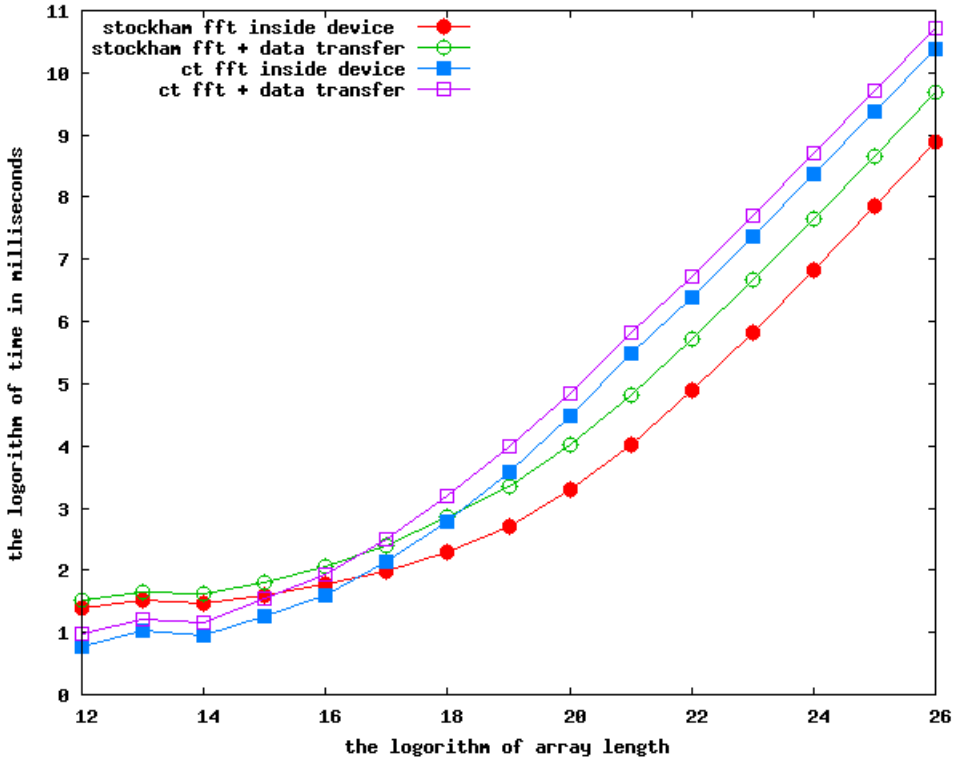

Figure 5.7: Comparison among GPU FFT implementations

Without considering the time spent in host-device data transfer, the speedup we achieve is about 37 for the FFT size 2^{26} (this speedup is about 21 if the data movement time is counted). As shown in Figure 5.7, where both the FFT size and the time are scaled by the base 2 logarithm, our Stockham FFT code is about 2 times faster than our Cooley-Tukey FFT code, mainly due to the jumped powers pre-computation.

5.6.3 Univariate polynomial multiplication over finite fields

As a direct application of fast Fourier transforms, we have implemented FFT based univariate polynomial multiplications over finite fields. Figure 5.8 compares the `modpn` FFT based polynomial multiplication against our GPU Stockham FFT-based one (the data transfer has been counted for the GPU code). The input two polynomials are randomly generated with the same given degree. When the degree is relatively large, the speedup we achieved is about 21 - 37, comparing to the `modpn` polynomial multiplication.

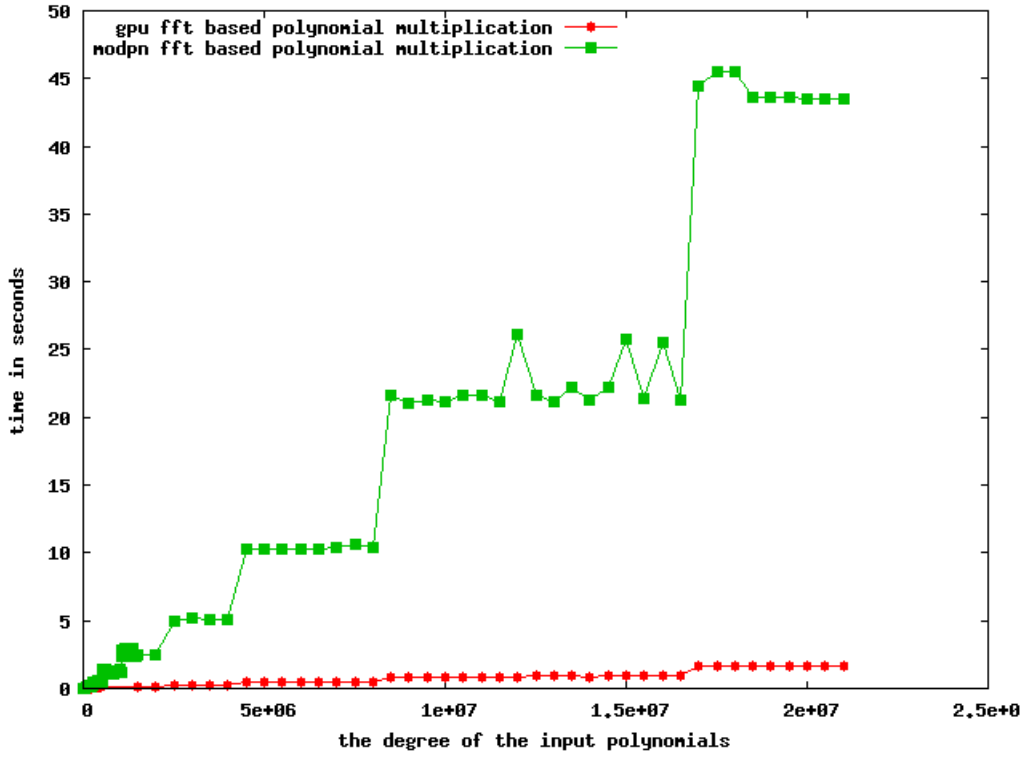

Figure 5.8: FFT-based dense polynomial multiplication on GPU and CPU

5.7 Summary

We have presented in detail various issues in implementing efficient fast Fourier transforms over finite fields on the GPU. Our experimental results show that the Stockham formula is well-suited for massively-threaded architectures. In particular, it avoids pre-computing extra powers of primitive roots of unity in a natural way, without downgrading the performance. Our implementation exhibits a significant performance improvement over a reference C implementation. For multiplying two dense univariate polynomials, we have achieved about a speedup of 30 with respect to the best code available to us. As our future work, we would like to implement multidimensional FFTs, and to revisit various modular algorithms in symbolic computation, like evaluation/interpolation based subresultant computations.

Chapter 6

Computing Subresultants

In Chapter 3, we have proposed a bottom-up algorithm for computing the regular GCDs of two polynomials modulo a regular chain. One of the key ingredients in the algorithm is that the computation of subresultants only needs to be computed *once* without modulo the regular chain, which brings the opportunity of using FFT based asymptotically fast algorithms. This chapter devotes to the computation of subresultants via an FFT based modular algorithm, and we report our GPU acceleration for this task.

6.1 Overview

Let $n \geq 1$ and let $P, Q \in \mathbf{k}[x_1, \ldots, x_{n+1}]$ be non-constant polynomials with the same main variable $y := x_{n+1}$ such that $p := \mathsf{deg}(P, y) \geq q := \mathsf{deg}(Q, y)$. We denote by $\mathbb{A}$ the ring $\mathbf{k}[x_1, \ldots, x_n]$ and by S_j the j-th subresultant of P, Q in $\mathbb{A}[y]$, for $0 \leq j < q$. For given positive integers $m_1, \ldots, m_n$, we call the following finite set Θ a *grid* in $\mathbf{k}^n$

$$\begin{aligned} \Theta &= \Theta_1 \times \Theta_2 \times \cdots \times \Theta_n \\ &= \{(u_1, \ldots, u_n) \mid u_i \in \Theta_i \text{ for each } i\} \end{aligned} \tag{6.1}$$

where Θ_i is a finite subset of $\mathbf{k}$ with size m_i for each i. If m_i is a power of 2 for each i, there exists a special type of grid, called *DFT grid*, where

$$\Theta_i = \{\omega_i^j \mid j = 0 \cdots m_i - 1\}$$

and ω_i is a m_i-th primitive root of unity for each i. The *format* of Θ is $(m_1, \ldots, m_n)$ and its *size* is the product $m_1 \cdots m_n$. We say the grid Θ is *valid* for a polynomial

$f \in \mathbf{k}[x_1, \ldots, x_n][y]$, if the leading coefficient of f in y does not vanish at any point in Θ.

The *evaluation-interpolation method* for computing subresultants S_j, proposed by Collins in [18], proceeds as follows.

($\mathbf{S}_1$) Compute an upper bound for the degree $\mathsf{deg}(S_0, x_i)$ and set m_i to it, for $i = 1 \cdots n$,

($\mathbf{S}_2$) Construct a grid Θ of format $(m_1, \ldots, m_n)$, valid for both P and Q,

($\mathbf{S}_3$) Evaluation: compute $P(u, y)$ and $Q(u, y)$ for each $u \in \Theta$,

($\mathbf{S}_4$) For each $u \in \Theta$, for each $0 \leq j < q$, compute the subresultants $S_j(u)$ of $P(u, y)$, $Q(u, y)$ in y,

($\mathbf{S}_5$) Interpolation: for each $0 \leq j < q$, construct the subresultant S_j from the image set $\{S_j(u) \mid u \in \Theta\}$.

For Step ($\mathbf{S}_1$), a well-known degree bound can be derived from the Sylvester matrix of P and Q, see the paper of Monagan [59] for detailed discussions. For Step ($\mathbf{S}_2$), let h be the product of the leading coefficients of P and Q in y, which is a nonzero polynomial in $\mathbf{k}[x_1, \ldots, x_n]$. Constructing a valid grid for h deterministically is difficult. In practice, one can generate a grid at random and check whether the grid is valid or not. Step ($\mathbf{S}_4$) is equivalent to computing the subresultant chains of m univariate polynomial pairs. A standard tool for doing this is Brown's subresultant algorithm [10].

Step ($\mathbf{S}_3$) and Step ($\mathbf{S}_5$) are instances of the so-called *multipoint evaluation* and *multipoint interpolation* problems, respectively. In general, these operations can be performed by means of subproduct tree techniques [34]. We do not analyze this point of view further since our focus is on DFT grids. If Θ is a DFT grid, then FFT-based multipoint evaluation and interpolation run in $\frac{3}{2}\mathsf{M}(m)\log(m)$ operations in $\mathbf{k}$. [1]

The approach presented in this chapter is summarized in Figure 6.1, where all the computations are converted into the bivariate case, by means of Kronecker's substitutions. This turns multivariate FFT computations into univariate ones, which simplifies both the analysis and the implementation. In practice, it is wiser to conduct large univariate FFTs (or large bivariate FFTs, see [68] for details) rather than multivariate FFTs with small sizes along one or more dimensions.

[1] In this chapter, $\mathsf{M}(m)$ denotes the number of arithmetic operations for computing the product of two univariate polynomials of degree less than m over a field. For those fields supporting FFTs, we have $\mathsf{M}(m) = O(m \log(m))$.

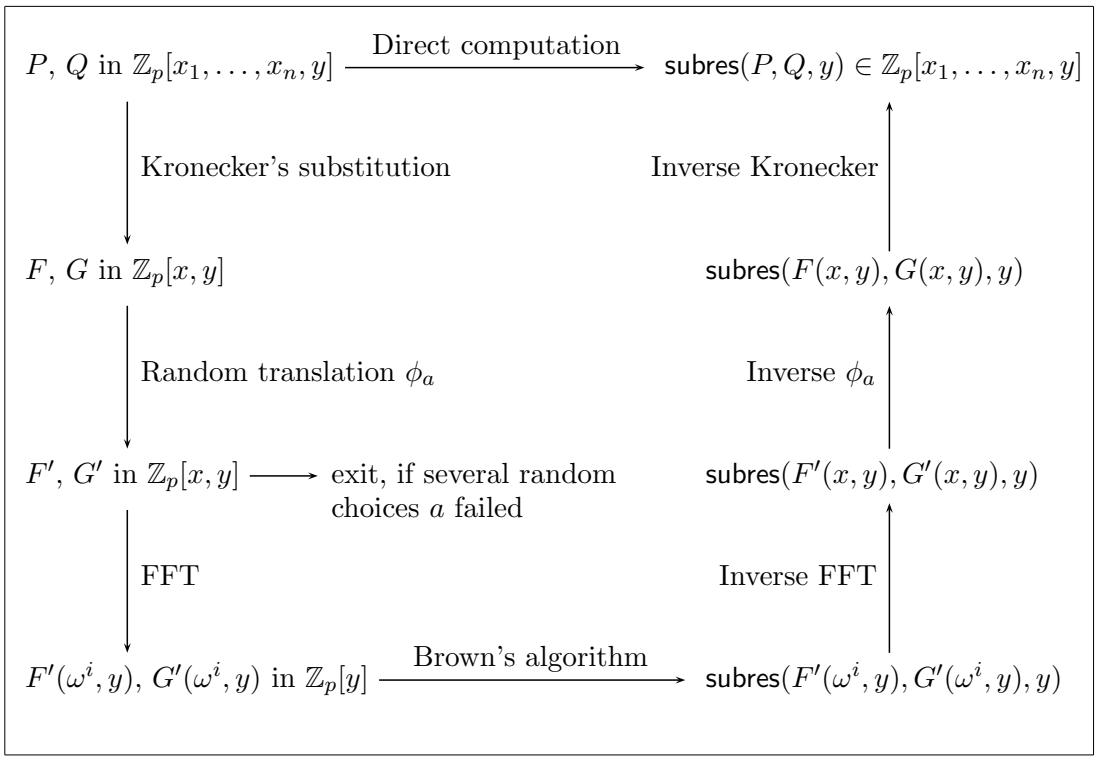

Figure 6.1: Compute the subresultant chain via a FFT based modular algorithm.

We organize this chapter as follows. Section 6.2 reviews the Kronecker substitutions. In Section 6.3, we recall properties of subresultants that are not covered in Section 3.2.3 of Chapter 3. In Section 6.4, we introduce and analyze the linear translation method for finding a valid DFT grid in the subresultant computations. Section 6.5 reviews Brown's subresultant algorithm for univariate polynomials. In Section 6.6, we analyze the over cost and memory consumption of building subresultant evaluation cube. In Section 6.7, we report our GPU implementation and experimental results on FFT based subresultant chain construction for bivariate and trivariate input.

6.2 Kronecker's substitution and its inverse

The Kronecker's substitution [34] is a well-known technique to transform a multivariate problem into the univariate one. In this section, we review some elementary properties of the Kronecker substitution and present an extension of it.

Definition 10. *Let k be a field and $f \in k[x_1, \ldots, x_n]$ such that $\mathsf{deg}(f, x_i) < e_i$ for*

$i = 1 \cdots n$. *The Kronecker's substitution with respect to* $(e_1, \ldots, e_n)$ *is defined as*

$$\begin{aligned}
&\delta : k[x_1, \ldots, x_n] \longrightarrow k[x], \\
&\delta(x_1) = x, \\
&\delta(x_i) = x^{e_1 e_2 \cdots e_{i-1}}, \quad i = 2 \cdots n \\
&\delta\left(\sum c_{a_1 \cdots a_n} x_1^{a_1} \cdots x_n^{a_n}\right) = \sum c_{a_1 \cdots a_n} \delta(x_1)^{a_1} \cdots \delta(x_n)^{a_n}.
\end{aligned} \tag{6.2}$$

That is, δ *directly substitutes each variable* x_i *by a predefined power of* x.

Proposition 11. *Let* $U = \{f \in k[x_1, \ldots, x_n] \mid \mathsf{deg}(f, x_i) < e_i, \text{ for } 1 \leq i \leq n\}$ *and let* $V = \{g \in k[x] \mid \mathsf{deg}(g) < e_1 \cdots e_n\}$. *The following properties hold*

(1) $\delta(m_1) = \delta(m_2) \iff m_1 = m_2$ *for monomials* $m_1, m_2 \in U$.

(2) $\delta(f + g) = \delta(f) + \delta(g)$ *for* $f, g \in U$.

(3) *Map* δ *gives a vector space isomorphism between* U *and* V. *This is to say, polynomials in* U *can be uniquely recovered from their image under* δ.

(4) $\delta(fg) = \delta(f)\delta(g)$ *if* $\mathsf{deg}(f, x_i) + \mathsf{deg}(g, x_i) < e_i$ *for* $i = 1 \cdots n$.

Proof. We first show that for all $i = 1 \cdots n - 1$ the following inequality holds

$$t_1 + t_2 e_1 + \cdots + t_i e_1 e_2 \cdots e_{i-1} < e_1 \cdots e_i, \tag{6.3}$$

whenever $t_i < e_i$ for $i = 1 \cdots n - 1$. We prove by induction on i. If $i = 1$, Inequality (6.3) holds since $t_1 < e_1$. For any $i > 1$, we have

$$\begin{aligned}
&t_1 + t_2 e_1 + \cdots + t_{i-1} e_1 e_2 \cdots e_{i-2} + t_i e_1 e_2 \cdots e_{i-1} \\
&< e_1 \cdots e_{i-1} + t_i e_1 e_2 \cdots e_{i-1} \quad \text{(by induction hypothesis)} \\
&\leq e_1 \cdots e_{i-1} e_i \quad \text{(since } t_i + 1 \leq e_i\text{)}.
\end{aligned}$$

For (1), let $m_1 = x_1^{a_1} \cdots x_n^{a_n} \in U$ and $m_2 = x_1^{b_1} \cdots x_n^{b_n} \in U$. Then we have

$$\begin{aligned}
\delta(m_1) = \delta(m_2) &\implies a_1 + \cdots + a_n e_1 e_2 \cdots e_{n-1} = b_1 + \cdots + b_n e_1 e_2 \cdots e_{n-1} \\
&\implies a_n = b_n \quad \text{(take the quotient by } e_1 \cdots e_n \text{ and use Inequality (6.3))}
\end{aligned}$$

Continuing in this way, we derive $a_i = b_i$ for all $i = 1 \cdots n$. (2) is a direct consequence of (1) and the definition of δ.

For (3), from Equation (6.2) and (2), δ gives an injective linear map from U to V. We deduce (3) from the fact that $\dim_{\mathbf{k}} U = \dim_{\mathbf{k}} V = e_1 \dots e_n$. For (4), if f is a term, that is, f can be written as cm for a monomial $m = x_1^{a_1} \cdots x_n^{a_n} \in U$ and $c \in \mathbf{k}$, then

$$\begin{aligned}
\delta(fg) &= \delta\left(cm \sum c_{b_1 b_2 \cdots b_n} x_1^{b_1} \cdots x_n^{b_n}\right) \\
&= \delta\left(\sum c c_{b_1 b_2 \cdots b_n} x_1^{a_1+b_1} \cdots x_n^{a_n+b_n}\right) \\
&= \sum c c_{b_1 b_2 \cdots b_n} \delta(x_1)^{a_1+b_1} \cdots \delta(x_n)^{a_n+b_n} \\
&= c\delta(x_1)^{a_1} \cdots \delta(x_n)^{a_n} \sum c_{b_1 b_2 \cdots b_n} \delta(x_1)^{b_1} \cdots \delta(x_n)^{b_n} \\
&= \delta(f)\delta(g).
\end{aligned}$$

For an arbitrary f, we write $f = \sum_i f_i$ with f_i being terms. By (2), we have

$$\delta(fg) = \delta\left(\sum_i f_i g\right) = \sum_i \delta(f_i g) = \sum_i \delta(f_i)\delta(g) = \left(\sum_i \delta(f_i)\right)\delta(g) = \delta(f)\delta(g).$$

This completes the proof. □

The above proof indicates a way to recover a polynomial in U from its image monomial by monomial. Each exponent a in the monomial x^a uniquely defines a sequence $(a_1, \dots a_n)$ as follows:

$$\begin{array}{rclcrcl}
a_n & = & a \text{ quo } e_1 \cdots e_{n-1} & & r_{n-1} & = & a \bmod e_1 \cdots e_{n-1} \\
a_{n-1} & = & r_{n-1} \text{ quo } e_1 \cdots e_{n-2} & & r_{n-2} & = & r_{n-1} \bmod e_1 \cdots e_{n-2} \\
\cdots & & & & & & \\
a_2 & = & r_2 \text{ quo } e_1 & & r_1 & = & r_2 \bmod e_1 \\
a_1 & = & r_1. & & & &
\end{array}$$

Then monomial $x_1^{a_1} \cdots x_n^{a_n}$ is the desired pre-image of x^a.

Example 15. *Let* $f = 1 + 2\,x_1 + 3\,x_2^2 + 4\,x_3^3 + 5\,x_1x_2 + 6\,x_1x_2x_3 \in \mathbb{Q}[x_1, x_2, x_3]$ *and let* $(e_1, e_2, e_3) = (2, 4, 4)$. *Then* δ *maps* (x_1, x_2, x_3) *to* (x, x^2, x^8) *and*

$$\delta(f) = h(x) = 1 + 2\,x + 3\,x^4 + 4\,x^{24} + 5\,x^3 + 6\,x^{11}.$$

The pre-image $\delta^{-1}(h)$ *of* h *is*

$$\begin{aligned}\delta^{-1}(h) &= 1 + 2\,\delta^{-1}(x) + 3\,\delta^{-1}(x^4) + 4\,\delta^{-1}(x^{24}) + 5\,\delta^{-1}(x^3) + 6\,\delta^{-1}(x^{11}) \\ &= 1 + 2\,x_1 + 3\,x_2^2 + 4\,x_3^3 + 5\,x_1x_2 + 6\,x_1x_2x_3.\end{aligned}$$

For instance, $\delta^{-1}(x^{11}) = x_1x_2x_3$ *is from the following calculations:*

$$\begin{array}{lcllcl} a_3 & = & 11 \text{ quo } 8 = 1 & \qquad r_2 & = & 11 \bmod 8 = 3 \\ a_2 & = & 3 \text{ quo } 2 = 1 & \qquad r_1 & = & 3 \bmod 2 = 1 \\ a_1 & = & 1. & & & \end{array}$$

The classic Kronecker substitution can be adapted to turn a multivariate into a bivariate one, which is also useful for some cases. This bivariate transformation is called *contraction* in [68] where it is used to optimize parallelism and cache complexity of dense multivariate polynomials in a multi-core implementation.

Definition 11. *Let* k *be a field and* $f \in k[x_1, \ldots, x_n]$ *such that* $\mathsf{deg}(f, x_i) < e_i$ *for* $i = 1 \cdots n$*. The bivariate Kronecker's substitution with respect to* $(e_1, \ldots, e_n)$ *and an index* $2 \leq j \leq n$ *is defined as*

$$\begin{aligned}&\delta_j : k[x_1, \ldots, x_n] \longrightarrow k[x, y], \\ &\quad \delta_j(x_1) = x, \\ &\quad \delta_j(x_i) = x^{e_1e_2\cdots e_{i-1}}, \quad i = 2 \cdots j-1 \\ &\quad \delta_j(x_j) = y, \\ &\quad \delta_j(x_i) = y^{e_je_{j+1}\cdots e_{i-1}}, \quad i = j+1 \cdots n \\ &\quad \delta_j\left(\sum c_{a_1\cdots a_n} x_1^{a_1} \cdots x_n^{a_n}\right) = \sum c_{a_1\cdots a_n} \delta_j(x_1)^{a_1} \cdots \delta_j(x_n)^{a_n}. \end{aligned} \tag{6.4}$$

Similar to Proposition 11, it is not hard to verify the following properties for bivariate Kronecker's substitutions δ_j.

Proposition 12. *Let* $U = \{f \in k[x_1, \ldots, x_n] \mid \mathsf{deg}(f, x_i) < e_i,\ \textit{for } 1 \leq i \leq n\}$ *and let*

$$V = \{g \in k[x, y] \mid \mathsf{deg}(g, x) < e_1 \cdots e_{j-1}, \mathsf{deg}(g, y) < e_j \cdots e_n\},$$

for some $2 \leq j \leq n$*. The following properties hold*

(1) $\delta_j(m_1) = \delta_j(m_2) \iff m_1 = m_2$ *for monomials* $m_1, m_2 \in U$*.*

(2) $\delta_j(f + g) = \delta_j(f) + \delta_j(g)$ *for* $f, g \in U$*.*

(3) *Map* δ_j *gives a vector space isomorphism between* U *and* V*.*

(4) $\delta_j(fg) = \delta_j(f)\delta_j(g)$ *if* $\mathsf{deg}(f, x_i) + \mathsf{deg}(g, x_i) < e_i$ *for* $i = 1 \cdots n$*.*

6.3 Resultants and subresultants

Resultant, as a powerful tool in the elimination theory, has been widely used in various triangular decomposition algorithms. The Triade algorithm developed by Moreno Maza [65] has been using it extensively. Let $P, Q \in R[y]$ be two non-zero polynomials of respective degree m and n such that $n, m > 0$. Suppose that

$$P = a_m y^m + \cdots + a_1 y + a_0 \quad \text{and} \quad Q = b_n y^n + \cdots + b_1 y + b_0.$$

Definition 12. *The Sylvester matrix of* P *and* Q *is a square matrix of order* $n + m$ *with coefficients in* R*, denoted by* $\mathsf{sylv}(P, Q, y)$ *and defined by*

$$\mathsf{sylv}(P,Q,y) = \begin{bmatrix}
a_m & a_{m-1} & \cdots & \cdots & \cdots & \cdots & a_0 & & & \\
 & a_m & a_{m-1} & \cdots & \cdots & \cdots & \cdots & a_0 & & \\
 & & \ddots & \ddots & & & & & \ddots & \\
 & & & a_m & a_{m-1} & \cdots & \cdots & \cdots & \cdots & a_0 \\
b_n & b_{n-1} & \cdots & b_1 & b_0 & & & & & \\
 & b_n & b_{n-1} & \cdots & b_1 & b_0 & & & & \\
 & & \ddots & \ddots & & & \ddots & & & \\
 & & & \ddots & \ddots & & & \ddots & & \\
 & & & & \ddots & \ddots & & & \ddots & \\
 & & & & & b_n & b_{n-1} & \cdots & \cdots & b_0
\end{bmatrix}$$

whose determinant, denoted by $\mathsf{res}(P, Q, y)$*, is called the resultant of* P *and* Q*.*

An important relation between the gcd and the resultant of P and Q is given by the following proposition, which tells in what condition that P, Q have a nontrivial common factor over an integral domain R, like the ring $\mathbb{Z}$ of integers. In particular, if R is a field, then P and Q have a common solution in R if and only if $\mathsf{res}(P, Q, y) = 0$.

Proposition 13 ([58])**.** *If* R *is a unique factorization domain (UFD), then* $\mathsf{gcd}(P, Q)$ *is non-constant in* $R[y]$ *if and only if* $\mathsf{res}(P, Q, y) = 0$ *in* R*.*

Example 16. *Let $P = ay^2 + by + c$ and let $Q = 2ay + b$ be the derivative of P w.r.t y. Then the Sylvester matrix of P and Q w.r.t y is*

$$S = \begin{bmatrix} a & b & c \\ 2a & b & 0 \\ 0 & 2a & b \end{bmatrix}$$

whose determinant is $\det(S) = a(4ac - b^2)$. *Whenever* $a \neq 0$, P *and* Q *have a common solution (or equivalently, $P = 0$ has a solution of multiplicity 2) if and only if the resultant* $\mathsf{res}(P, Q, y)$ *is zero.*

The notion of subresultant is a refinement of resultant. Each subresultant of P and Q is a polynomial in y whose coefficients are minors of its Sylvester matrix.

Definition 13 (Determinantal polynomial)**.** *Let $m \leq n$ be positive integers. Let M be a $m \times n$ matrix with coefficients in a commutative ring R. Let M_i be the square submatrix of M consisting of the first $m-1$ columns of M and the i-th column of M, for $i = m \cdots n$; let* $\det M_i$ *be the determinant of M_i. The* determinantal polynomial *of M, denote by* $\mathsf{dpol}(M)$*, is a polynomial in $R[y]$, given by*

$$\mathsf{dpol}(M) = \det M_m y^{n-m} + \det M_{m+1} y^{n-m-1} + \cdots + \det M_n.$$

If $\mathsf{dpol}(M)$ is nonzero then its degree is at most $n - m$. Let $P_1, \ldots, P_m$ be polynomials of $R[y]$ of degree less than n. We denote by $\mathsf{mat}(P_1, \ldots, P_m)$ the $m \times n$ matrix whose i-th row contains the coefficients of P_i, sorted in order of decreasing degree, and such that P_i is treated as a polynomial of degree $n - 1$. We denote by $\mathsf{dpol}(P_1, \ldots, P_m)$ the determinantal polynomial of $\mathsf{mat}(P_1, \ldots, P_m)$.

Example 17. *Let $n = 4$, $m = 2$, $P_1 = a_3y^3 + a_2y^2 + a_1y + a_0$ and $P_2 = b_2y^2 + b_1y + b_0$. Then*

$$\mathsf{mat}(P_1, P_2) = \begin{bmatrix} a_3 & a_2 & a_1 & a_0 \\ 0 & b_2 & b_1 & b_0 \end{bmatrix}, M_2 = \begin{bmatrix} a_3 & a_2 \\ 0 & b_2 \end{bmatrix},$$
$$M_3 = \begin{bmatrix} a_3 & a_1 \\ 0 & b_1 \end{bmatrix}, \text{ and } M_4 = \begin{bmatrix} a_3 & a_0 \\ 0 & b_0 \end{bmatrix}.$$

Consequently, we have $\mathsf{dpol}(P_1, P_2) = a_3b_2y^2 + a_3b_1y + a_3b_0$.

Definition 14. *Let $P, Q \in R[y]$ be non-constant polynomials of respective degrees p, q with $q \leq p$. Let k be an integer with $0 \leq k < q$. Then the k-th* subresultant *of P and*

Q, denoted by $\mathsf{subres}_k(P,Q,y)$*, is*

$$\mathsf{subres}_k(P,Q,y) = \mathsf{dpol}(y^{p-k-1}P, y^{p-k-2}P, \ldots, P, y^{q-k-1}Q, \ldots, Q).$$

Observe that if $\mathsf{subres}_k(P,Q,y)$ *is not zero then its degree is at most* k*. When* $\mathsf{subres}_k(P,Q,y)$ *has degree* k*, it is said to be* regular*; when* $\mathsf{subres}_k(P,Q,y) \neq 0$ *and* $\mathsf{deg}(\mathsf{subres}_k(P,Q,y)) < d$*,* $\mathsf{subres}_k(P,Q,y)$ *is said to be* defective*.*

It is easy to show that $\mathsf{subres}_0(P,Q,y)$ is $\mathsf{res}(P,Q,y)$, the resultant of P and Q.

Example 18. *Let* $P = b_2y^2 + b_1y + b_0$ *and* $Q = a_3y^3 + a_2y^2 + a_1y + a_0$*. Then*

$$\begin{aligned}\mathsf{subres}_0(P,Q,y) = \mathsf{dpol}(y^2P, yP, P, yQ, Q) &= \mathsf{dpol}\begin{bmatrix} b_2 & b_1 & b_0 & & \\ & b_2 & b_1 & b_0 & \\ & & b_2 & b_1 & b_0 \\ a_3 & a_2 & a_1 & a_0 & \\ & a_3 & a_2 & a_1 & a_0 \end{bmatrix} \\ &= b_2a_2^2b_0^2 - 2b_2^2a_2b_0a_0 - a_2b_0^2a_3b_1 + b_2^3a_0^2 + 3b_2a_0a_3b_1b_0 \\ &\quad - b_1b_2a_1a_2b_0 - b_1b_2^2a_1a_0 + b_1^2a_1a_3b_0 + b_2a_2b_1^2a_0 \\ &\quad - a_3b_1^3a_0 + b_0b_2^2a_1^2 - 2b_2a_1a_3b_0^2 + a_3^2b_0^3\end{aligned}$$

and

$$\begin{aligned}\mathsf{subres}_1(P,Q,y) = \mathsf{dpol}(yP,P,Q) &= \mathsf{dpol}\begin{bmatrix} b_2 & b_1 & b_0 & \\ & b_2 & b_1 & b_0 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \\ &= (b_2^2a_1 - b_2a_3b_0 - b_2a_2b_1 + a_3b_1^2)\,y - b_2a_2b_0 + b_2^2a_0 + a_3b_1b_0.\end{aligned}$$

In particular, when $P = y(y-3) = y^2 - 3y$ *and* $Q = y(y-1)(y+1) = y^3 - y^2$*, we have* $\mathsf{subres}_0(P,Q) = 0$ *and* $\mathsf{subres}_1(P,Q) = 6y$*, which in fact reflects* $\mathsf{gcd}(P,Q) = y$*.*

Proposition 14. *Let* R *be a UFD and* P, Q *be polynomials in* $R[y]$ *with degrees* p *and* q*. If for some* $0 < k < \min(p,q)$*,* $\mathsf{subres}_k(P,Q,y) \neq 0$ *and* $\mathsf{subres}_i(P,Q,y) = 0$ *for all* $i < k$*, then we have* $\mathsf{deg}(\mathsf{gcd}(P,Q)) = k$*. In fact,* $\mathsf{subres}_k(P,Q,y)$ *is similar to* $\mathsf{gcd}(P,Q)$ *in the sense that there exist nonzero constants* α *and* β *such that* $\alpha\,\mathsf{gcd}(P,Q) = \beta\,\mathsf{subres}_k(P,Q)$*.*

According to the above proposition, $\mathsf{subres}_k(P,Q,y)$ is a regular subresultant, and we usually call it the *last subresultant* of P, Q, which is in fact the last nonzero

subresultant. This terminology is used in the `PRS` package of Lionel Ducos in AXIOM and in the `RegularChains` library in MAPLE.

In what follows, we review some known bounds on the resultants. They are of particular importance for computing subresultants using modular methods. We focus on the degree estimates on the resultant of two multivariate polynomials in the polynomial ring $\mathbf{k}[x_1, \ldots, x_n, y]$, where $\mathbf{k}$ is a field.

Proposition 15. *Let $f, g \in \mathbf{k}[x_1, \ldots, x_n, y]$. When writing $d_y = \mathsf{deg}(f, y)$, $d'_y = \mathsf{deg}(g, y)$, $d_i = \mathsf{deg}(f, x_i)$, and $d'_i = \mathsf{deg}(g, x_i)$ for $i = 1 \cdots n$, we have*

$$\mathsf{deg}(\mathsf{res}(f, g, y), x_i) \leq d'_y d_i + d_y d'_i, \tag{6.5}$$

for each $i = 1 \cdots n$.

Proof. According to the definition, the resultant of f and g in y is the determinant of its Sylvester matrix, which can be expanded as a sum of $(d_y + d'_y)!$ terms. Each nonzero term has d'_y factors from the coefficients of f, and d_y factors from coefficients of g. Thus in each term, the partial degree in x_i is at most $d'_y d_i + d_y d'_i$, which implies the claim. □

In [59], the author refined Proposition 15 by expanding the Sylvester matrix S of f and g in y in a different way. Define

$$\mu_{ir} = \max_{1 \leq s \leq d_y + d'_y} \mathsf{deg}(S_{rs}, x_i) \quad \text{and} \quad \nu_{is} = \max_{1 \leq r \leq d_y + d'_y} \mathsf{deg}(S_{rs}, x_i)$$

for $i = 1 \cdots n$. Expanding S along the columns gives the following row bound

$$\mathsf{deg}(\mathsf{res}(f, g, y), x_i) \leq \sum_{r=1}^{d_y + d'_y} \mu_{ir}. \tag{6.6}$$

Expanding S along the rows gives the following column bound

$$\mathsf{deg}(\mathsf{res}(f, g, y), x_i) \leq \sum_{s=1}^{d_y + d'_y} \nu_{is}. \tag{6.7}$$

For the bivariate case, there is a special bound on the degree of the resultant, which sometimes gives a tighter estimate.

Proposition 16 (Bézout Bound)**.** *Let $f, g \in \mathbf{k}[x, y]$. Then*

$$\mathsf{deg}(\mathsf{res}(f, g, y), x) \leq \mathsf{deg}(f)\mathsf{deg}(g), \tag{6.8}$$

where deg *is the total degree.*

Proof. This is a direct consequence of the Bézout Theorem, which says that the number of roots of $\mathsf{res}(f, g, y)$ is bounded by $\mathsf{deg}(f)\mathsf{deg}(g)$. □

Example 19. *Consider two polynomials f and g over $\mathbb{Q}[x, y]$ defined as*

$$f = x^2y - x^2 + 6\,y - 6 - xy^2 - x,$$
$$g = xy^2 - y^2 + 6\,x - 6 - x^2y - y.$$

The Sylvester matrix of f and g w.r.t y is

$$S = \begin{bmatrix} -x & x^2+6 & -x^2-6-x & 0 \\ 0 & -x & x^2+6 & -x^2-6-x \\ x-1 & -x^2-1 & 6\,x-6 & 0 \\ 0 & x-1 & -x^2-1 & 6\,x-6 \end{bmatrix}.$$

Let $r(x) = \mathsf{res}(f, g, y)$. Then the Sylvester bound in Equation (6.5) gives $\mathsf{deg}(r, x) \leq 2\times 2+2\times 2 = 8$, the row bound in Equation (6.6) gives $\mathsf{deg}(r, x) \leq 2+2+2+2 = 8$, the column bound in Equation (6.7) gives $\mathsf{deg}(r, x) \leq 1+2+2+2 = 7$, and the Bézout bound in Equation (6.8) gives $\mathsf{deg}(r, x) \leq 3 \times 3 = 9$. Hence, the tightest one is the column bound $\mathsf{deg}(r, x) \leq 7$. In fact, we have

$$r(x) = 2\,x^6 - 22\,x^5 + 102\,x^4 - 274\,x^3 + 488\,x^2 - 552\,x + 288,$$

which has degree 6.

6.4 Finding a valid DFT grid

Let P and Q be polynomials in $\mathbf{k}[\boldsymbol{x}, y]$, where $\boldsymbol{x}$ stands for $x_1, \ldots, x_n$,. We assume that $p = \mathsf{deg}(P, y) \geq q = \mathsf{deg}(Q, y) > 0$ holds. We write P and Q as

$$P = \sum_{i=0}^{p} a_i\,y^i \quad \text{and} \quad Q = \sum_{j=0}^{q} b_j\,y^j,$$

where a_i's and b_j's are polynomials in $x_1, \ldots, x_n$. Our goal is to compute the subresultant sequence $S_{e-1}, \ldots, S_1, S_0$ of P and Q in y, where $S_k = \mathsf{subres}_k(P, Q, y)$.

As presented in Figure 6.1, where $n = 1$ is assumed, the first step is to evaluate P and Q at a DFT grid by means of fast Fourier transforms. The FFT sizes m_i are determined by the degree in x_i of the resultant $S_0 = \mathsf{res}(P, Q, y)$. According to Proposition 15, Equation (6.7), Equation (6.6) and Proposition 6.8, we calculate an estimate on the degree bound D of S_0 in x_i, for all i.

There are other practical issues to be elaborated. For instance, assuming $n = 1$, if one of the leading coefficients has a factor $x_1 - 1$, then we can never find a valid DFT grid, since the DFT grid is of the form $[1, \omega_1, \ldots, \omega_1^{m_1-1}]$, in which 1 cancels one leading coefficient. More generally, it is possible that one of the leading coefficient vanishes at a power of ω_1. In this case, inverse FFTs can not be applied to recover subresultants correctly from specialized images. The following example illustrates the problem and shows how to overcome it by means of linear translations.

Example 20. *Let* $\mathbf{k} = \mathbb{Z}/p\mathbb{Z}$ *with* $p = 97$ *and consider bivariate polynomials*

$$F = (x-1)y^2 + (-x^2-1)y + (6\,x - 6) \quad \textit{and} \quad G = -xy^2 + (x^2+6)y + (-x^2 - x - 6).$$

The degree bound (in x*) of subresultants is* $D = 7$*, and hence we could set the FFT size to* $m = 8$*. Since the leading coefficient of* F *is* $x - 1$*, there is no valid DFT grid for* F *and* G*. We transform* F *and* G *by a "random" translation* $\phi : x \mapsto x + 21$*, and denote by* $F' = \phi(F)$ *and* $G' = \phi(G)$*:*

$$\begin{aligned} F' &= (x + 20)\,y^2 + \left(33 + 55\,x + 96\,x^2\right) y + 6\,x + 23 \\ G' &= (96\,x + 76)\,y^2 + \left(59 + x^2 + 42\,x\right) y + 96\,x^2 + 54\,x + 17 \end{aligned}$$

We have a 8*-th primitive root of unity* $\omega = 33$*, which defines an DFT grid*

$$\Theta = [1, \omega, \ldots, \omega^7] = [1, 33, 22, 47, 96, 64, 75, 50].$$

The product H *of the leading coefficients of* F' *and* G' *is* $96\,(x+21)\,(x+20)$*. Evaluating* H *at* Θ *gives* $[23, 48, 37, 3, 8, 38, 95]$*, which implies that* Θ *is a valid DFT grid for* F' *and* G'*.*

In what follows, we formally define linear translations and analyze in which case it is possible to overcome this problem. First, the specialization property of subresul-

tants as stated in Corollary 6 says that one can recover the subresultants of F, G in y from the ones of F', G' in y.

Theorem 10 (Subresultants under specialization)**.** *Let A and B be commutative rings with unity, and $\phi : A \to B$ be a ring homomorphism*[2]*. Let F and G be univariate polynomials in $A[y]$ with degrees d and e, respectively. Assuming that $d \geq e$ and $\phi(\mathsf{lc}(F)\mathsf{lc}(G)) \neq 0$ hold. Then we have the identity*

$$\phi(\mathsf{subres}_i(F, G, y)) = \mathsf{subres}_i(\phi(F), \phi(G), y), \quad \textit{for all } 0 \leq i < e. \tag{6.9}$$

Proof. See the proof from [58]. □

Example 21. *For any positive integer m, the map $\phi : \mathbb{Z} \to \mathbb{Z}/m\mathbb{Z}$, $k \mapsto k \bmod m$, is a ring homomorphism. For any commutative ring A with unity and $a \in A$, the translation map $\phi : A[x] \to A[x]$, $f(x) \mapsto f(x + a)$ is a homomorphism from $A[x]$ to itself.*

Definition 15. *For any $a = (a_1, \ldots, a_n) \in \mathbf{k}^n$, define the translation map ϕ_a with respect to a as follows*

$$\begin{aligned} &\phi_a : \mathbf{k}[\boldsymbol{x}, y] \longrightarrow \mathbf{k}[\boldsymbol{x}, y] \\ &f(x_1, \ldots, x_n, y) \mapsto f(x_1 + a_1, \ldots, x_n + a_n, y). \end{aligned}$$

It is easy to check that the inverse translation of ϕ_a is ϕ_{-a} defined by $-a = (-a_1, \ldots, -a_n)$.

Corollary 6. *Let P, Q be polynomials in $\mathbf{k}[\boldsymbol{x}, y]$ with positive degrees in y. Let ϕ_a be a translation map such that neither $\phi_a(\mathsf{lc}(P, y))$ nor $\phi_a(\mathsf{lc}(Q, y))$ is zero. Then the following identity holds*

$$\phi_{-a}(\mathsf{subres}_j(\phi_a(P), \phi_a(Q), y)) = \mathsf{subres}_j(P, Q, y), \quad \textit{for all } 0 \leq j < e, \tag{6.10}$$

where $e = \min(\mathsf{deg}(P, y), \mathsf{deg}(Q, y))$.

Proof. A translation map ϕ_a is a ring homomorphism from $\mathbf{k}[\boldsymbol{x}]$ to itself. □

In the following two subsections, we analyze when a translation ϕ_a permits a valid DFT grid of size m for $\phi_a(P)$ and $\phi_a(Q)$. We restrict ourselves to the case where $\mathbf{k}$ is a finite field and $n = 1$ or $n = 2$.

[2] A ring homomorphism is a function from A to B such that $\phi(1_A) = 1_B$, $\phi(a + b) = \phi(a) + \phi(b)$ and $\phi(ab) = \phi(a)\phi(b)$.

6.4.1 Translations over a finite field with $n = 1$

Let $\mathbf{k} = \mathbb{Z}/p\mathbb{Z}$ for some prime $p > 2$. Write p in the form $c\,2^k + 1$ with c being odd, [3] and let $f(x) \in \mathbf{k}[x]$ be a monic univariate polynomial of degree at most d.

Given a grid size $m \leq 2^k$ and an m-th primitive root of unity $\omega \in \mathbf{k}$, we bound the number of translations ϕ_a such that the DFT grid $[1, \ldots, \omega^{m-1}]$ is valid for $\phi_a(f)$. Equivalently, we measure the size of the set

$$\mathcal{A}(f, m, \omega) = \{a \in \mathbf{k} \mid \phi_a(f) = f(x + a) \text{ does not vanish at any power of } \omega\}. \quad (6.11)$$

Since ω is an m-th primitive root of unity, $\omega^i \equiv \omega^{i \bmod m} \bmod p$ holds for all positive integers i. The equality

$$\prod_{i=0}^{m-1} (x - \omega^i) = x^m - 1$$

(which is easily proved by evaluating both sides at each power of ω) implies that the set $\mathcal{A}$ can be reformulated as

$$\mathcal{A}(f, m, \omega) = \{a \in \mathbf{k} \mid \mathsf{gcd}\,(f(x + a), x^m - 1) = 1\}. \quad (6.12)$$

An important implication of Equation (6.12) is: the set $\mathcal{A}(f, m, \omega)$ is independent of the choice of ω. Consequently, we also write $\mathcal{A}(f, m, \omega)$ as $\mathcal{A}(f, m)$.

Without computing the gcd, condition $\mathsf{gcd}\,(f(x + a), x^m - 1) = 1$ can be checked via FFT computations. The following simple subroutine `isGoodShift`(f, m, a) checks if $a \in \mathcal{A}(f, m)$ holds or not, in the sense that $f(x + a)$ vanishes at no powers of any m-th primitive roots of unity.

Algorithm 6: `isGoodShift`(f, m, a)

Input : Polynomial f in $\mathbb{Z}/p\mathbb{Z}[x]$, the DFT grid size m, and $a \in \mathbb{Z}_p$
Output : true if $a \in \mathcal{A}(f, m)$, false otherwise

1 Compute $\hat{f}(x) \leftarrow f(x + a)$
2 Compute $\mathbf{v} \leftarrow \mathrm{DFT}_m(\hat{f}, \omega)$ for some m-th primitive root of unity ω
3 **if** $\mathbf{v}[i] = 0$ *for some* $0 \leq i < m$ **then return** *false*
4 **return** *true*

[3] In this form k is called the Fourier degree of p and 2^k is called the Fourier size. Over $\mathbb{Z}/p\mathbb{Z}$, the longest data length for a fast Fourier transform is the Fourier size of p, since there is no primitive root of unity with order more than 2^k.

Proposition 17. *The Algorithm 6 runs in $O(d^2+m\log m)$ operations in $\mathbb{Z}/p\mathbb{Z}$, where d is the degree of f.*

Proof. The cost to compute $\hat{f}$ is in $O(d^2)$. According to the Horner's rule we can write $\hat{f}(x)$ as

$$f(x+a) = f_0 + (x+a)(f_1 + \cdots + (x+a)(f_{d-1} + f_d(x+a))\cdots),$$

where $f(x) = \sum_{i=0}^{d} f_i\, x^i$. Denote by $C(d)$ the additions and multiplications by a performed in the above Horner's formula. We have the following recurrence relation

$$C(d) = C(d-1) + 2d, \quad C(1) = 3,$$

which gives $C(d) \in O(d^2)$. The total cost in step 2 and 3 is in $O(m\log m + m) = O(m\log m)$. Hence the total cost is in $O(d^2 + m\log m)$ as desired. □

Remark 6. *Note that computing $\hat{f} = f(x+a)$ is also known as Taylor shift. The fast algorithms in [81] can achieve $\mathsf{M}(d) = O(d\log d)$ operations in $\mathbf{k}$. Therefore, the overall cost of Algorithm 6 can be improved to $O(d\log d + m\log m)$.*

The following lemma characterizes when a is an element in $\mathcal{A}(f,m)$, which can be used to bound its cardinality.

Lemma 20. *The condition $a \in \mathcal{A}(f,m)$ holds if and only if $\lambda(a) \neq 0$ where $\lambda(u)$ is a univariate polynomial in $\mathbb{Z}/p\mathbb{Z}[u]$ defined by*

$$\lambda(u) = \mathsf{res}(f(x+u), x^m - 1, x).$$

Moreover, the degree of $\lambda(u)$ is $m \times \mathsf{deg}(f)$.

Proof. By definition,

$$\begin{aligned} a \in \mathcal{A}(f,m) &\iff \mathsf{gcd}(f(x+a), x^m - 1) = 1 \\ &\iff \mathsf{res}(f(x+a), x^m - 1, x) \neq 0 \\ &\iff \lambda(a) \neq 0. \end{aligned}$$

Since $x^m - 1 = \prod_{i=0}^{m-1}(x - \omega^i)$ holds for an m-th primitive root of unity, we have

$$\begin{aligned}\lambda(u) = \mathsf{res}(f(x+u), x^m - 1, x) &= \prod_{i=0}^{m-1} \mathsf{res}(f(x+u), x - \omega^i, x) \\ &= \prod_{i=0}^{m-1} f(u + \omega^i).\end{aligned}$$

For each i, $f(u+\omega^i)$ has the same degree as f. Hence $\mathsf{deg}(\lambda) = m \times \mathsf{deg}(f)$ holds. □

Example 22. *Let $p = 97$, $\mathbf{k} = \mathbb{Z}/p\mathbb{Z}$, $m = 16$ and $f(x) = x^4 + 45\,x^3 + 82\,x^2 + 92\,x + 71$. Then the set $\mathcal{A}(f, m)$ is*

$$\left\{ \begin{array}{l} 1, 3, 4, 5, 6, 7, 8, 10, 11, 14, 16, 17, 18, 20, 21, 22, 24, 25, 26, 27, 30, 31, 32, 33, \\ 36, 37, 38, 39, 41, 42, 43, 45, 46, 47, 49, 52, 53, 55, 56, 57, 58, 59, 60, 62, 64, \\ 66, 67, 68, 69, 72, 73, 75, 77, 78, 79, 81, 82, 83, 85, 87, 88, 91, 92, 93, 94, 96 \end{array} \right\}.$$

Its complement is

$$\left\{ \begin{array}{l} 0, 2, 9, 12, 13, 15, 19, 23, 28, 29, 34, 35, 40, 44, 48, 50, \\ 51, 54, 61, 63, 65, 70, 71, 74, 76, 80, 84, 86, 89, 90, 95 \end{array} \right\}.$$

The cardinality of $\mathcal{A}(f, m)$ is 66.

In Example 22, if we pick an $a \in \{0, \ldots, p-1\}$ uniformly at random, then the probability of a being a "good" translation for f is approximately 0.68. The following proposition gives a condition in which such a probability is at least 1/2.

Proposition 18. *If the degree of f is at most $\frac{p}{2\,m}$, then the cardinality of $\mathcal{A}(f, m)$ is at least $\frac{p}{2}$.*

Proof. According to Lemma 20, the degree of $\lambda(u) = \mathsf{res}(f(x+u), x^m - 1)$ is

$$m \times \mathsf{deg}(f) \leq m \times \frac{p}{2\,m} = \frac{p}{2}.$$

Hence the number of solutions of $\lambda(u)$ in $\mathbb{Z}/p\mathbb{Z}$ is at most $\frac{p}{2}$, and consequently the cardinality of $\mathcal{A}(f, m)$ is at least $\frac{p}{2}$. □

For example, let $p = 469762049 = 7 \times 2^{26} + 1$ be a large machine prime. If $m = 2^{20}$ and $\mathsf{deg}(f) \leq 448$, then at least half of a in $\mathbb{Z}/p\mathbb{Z}$ give "good" translations for f. This bound is already quite practical. Note that the bound in Proposition 18 is pessimistic, since it ignores the structure of f at all. In Example 22, $f =$

$(x-1)(x^3+46\,x^2+31\,x+26)$ is the product of $x-1$ and a randomly generated degree 3 monic polynomial. The cardinality bound is $\#\mathcal{A}(f,m) \geq p - m \times \mathsf{deg}(f) = 33$. In fact, its cardinality is 66.

6.4.2 Translations over a finite field with $n = 2$

Similarly, we define the set of "good" translations of a bivariate polynomial $f(x,y)$ as follows

$$\mathcal{A}_2(f, m_1, m_2, \omega_1, \omega_2) = \left\{ (a,b) \in \mathbf{k}^2 \;\middle|\; \begin{array}{l} f(x+a, y+b) \text{ does not vanish at} \\ \text{any point } (\omega_1^i, \omega_2^j) \text{ for } i,j \geq 0 \end{array} \right\}, \quad (6.13)$$

where ω_i is m_i-th primitive root of unity. It turns out that the polynomial

$$\begin{aligned} \lambda(u,v) &= \mathsf{res}(\mathsf{res}(f(x+u, y+v), x^{m_1} - 1, x), y^{m_2} - 1, y) \\ &= \mathsf{res}\left(\prod_{i=0}^{m_1-1} f(\omega_1^i + u, y+v), y^{m_2} - 1, y \right) \\ &= \prod_{i=0}^{m_1-1} \prod_{j=0}^{m_2-1} f(u + \omega_1^i, v + \omega_2^j) \end{aligned}$$

decides whether a pair $(a,b) \in \mathbf{k}^2$ belongs to $\mathcal{A}_2(f, m_1, m_2, \omega_1, \omega_2)$ or not. Again, Definition 6.13 is independent of the choice of ω_1 and ω_2. The partial degree of $\lambda(u,v)$ in u (or v) is bounded by $m_1 \times m_2 \times \mathsf{deg}(f,x)$ (or $m_1 \times m_2 \times \mathsf{deg}(f,y)$), respectively. Similar to Lemma 20, we have

Proposition 19. *The condition $(a,b) \in \mathcal{A}_2(f, m_1, m_2)$ holds if and only if $\lambda(a,b) \neq 0$. The number of solutions of $\lambda(u,v)$ is at most $m_1\, m_2\, p\, \min(\mathsf{deg}(f,x), \mathsf{deg}(f,y))$.*

Proof. For an arbitrary choice $v = v_0 \in \mathbb{Z}_p$, $\lambda(u, v_0)$ is an univariate polynomial in u with degree at most $m_1\, m_2\, \mathsf{deg}(f,x)$, which implies that the number of solutions of $\lambda(u,v)$ in $\mathbb{Z}_p \times \mathbb{Z}_p$ is at most $m_1\, m_2\, p\, \mathsf{deg}(f,x)$. The above argument also holds if we switch u and v, and we prove the claim. □

In practice, we choose a sequence of pairs $a = (a_1, a_2) \in \mathbf{k}^2$ uniformly at random, and check whether each a belongs to $\mathcal{A}_2(f, m_1, m_2)$ or not. If several consecutive trials fail (say 5 to 10), it is most likely that the size $m = m_1 m_2$ is too big to the field characteristic p.

6.5 Brown's subresultant algorithm

To compute the GCD of two univariate polynomials over a domain, the polynomial remainder sequence (PRS) derived from the Euclidean algorithm suffers from the phenomenon of explosive coefficient growth, which fortunately is not inherent to the problem. The key to controlling coefficient growth is the discovery of subresultants, where each subresultant is in fact proportional to a polynomial in the PRS, [10, 11, 17]. The specialization property of subresultants makes it possible to apply modular techniques, while this property does not hold for a PRS.

In Algorithm 7, we list the subresultant algorithm of Brown [10] for two univariate polynomials over a field. This algorithm preallocates space for all subresultants and initialize them to zero. Then, this algorithm writes the nonzero subresultants, until it exists from the loop or detects termination at Line 7. The loop condition checks whether all remaining subresultants are zero or not, and Line 7 checks whether all subresultants are computed (the last one has index 0).

In total, there are at most $\mathsf{deg}(G)$ iterations from Line 3 to Line 8, since each iteration produces at least one new subresultant. The loop invariant is

$$A \text{ is a regular subresultant with } A = S_d,\ B = S_{d-1}, \text{ and } \delta \geq 1.\ ^{4}$$

At Line 8, the algorithm advances one step by replacing B with a proper multiple of $\mathsf{prem}(A, -B, x)$ and replacing A with S_e. If $\delta = 1$ at Line 6, then we have $S_{d-1} = S_e$ which implies that no new subresultant has been produced in this step.

Algorithm 7: Brown's subresultant algorithm

Input : polynomials $F, G \in \mathbf{k}[x]$ such that $\mathsf{deg}(F) \geq \mathsf{deg}(G) > 0$
Output : the subresultant chain of F and G

1 $S_i \leftarrow 0$ for $0 \leq i < \mathsf{deg}(G)$
2 $B \leftarrow \mathsf{prem}(F, -G, x)$, $A \leftarrow G$, $\alpha \leftarrow \mathsf{deg}(F) - \mathsf{deg}(G)$
3 **while** $B \neq 0$ **do**
4 $\quad d \leftarrow \mathsf{deg}(A)$, $e \leftarrow \mathsf{deg}(B)$, $\delta \leftarrow d - e$
5 $\quad S_{d-1} \leftarrow B$
6 $\quad S_e \leftarrow \mathsf{lc}(A)^{\alpha(1-\delta)}\mathsf{lc}(B)^{\delta-1}\, B$
7 $\quad$ **if** $e = 0$ **then break**
8 $\quad B \leftarrow \mathsf{lc}(A)^{-\alpha\delta-1}\mathsf{prem}(A, -B, x)$, $A \leftarrow S_e$, $\alpha \leftarrow 1$
9 **return** S_i *for* $0 \leq i < \mathsf{deg}(G)$

[4] For convenience, we regard the input polynomial G as a regular subresultant of index $\mathsf{deg}(G)$. Subresultant S_{d-1} might not be a regular subresultant, i.e, $\delta = d - e > 1$.

Example 23. *Consider univariate polynomials*

$$F = 5\,x^5 + 4\,x^4 + 2\,x^2 + 3\,x^3 + x \quad \textit{and} \quad G = 9\,x^4 + 7\,x^3 + 5\,x^2 + 3\,x + 1$$

in $\mathbb{Z}/17\mathbb{Z}[x]$. *The first pseudo-division (Line* 2*) produces*

$$S_3 = \mathsf{prem}(F, -G, x) = 11\,x^3 + 5\,x^2 + 16\,x + 16,$$

which is a regular subresultant, i.e. $\delta = d - e = 4 - 3 = 1$. *Thus Line* 6 *gets skipped. The second pseudo-division (Line* 8*) produces*

$$S_2 = \mathsf{lc}(G)^{-2}\mathsf{prem}(G, -S_3, x) = 4\,x,$$

which is a defective subresultant, i.e. $\delta = d - e = 3 - 1 = 2$. *Thus Line* 6 *computes the regular subresultant* S_1 *associated to* S_2

$$S_1 = \mathsf{lc}(S_3)^{-1}\mathsf{lc}(S_2)S_2 = 3\,x.$$

The third pseudo-division (Line 8*) produces*

$$S_0 = \mathsf{lc}(S_3)^{-3}\mathsf{prem}(S_3, -S_2, x) = 6,$$

which is the last subresultant of F *and* G. *The subresultant chain of* F *and* G *is*

$$\left|\begin{array}{lcl} S_3 & = & 11\,x^3 + 5\,x^2 + 16\,x + 16 \\ S_2 & = & 4\,x \\ S_1 & = & 3\,x \\ S_0 & = & 6 \end{array}\right.$$

in which S_3, S_1, S_0 *are regular and* S_2 *is defective.*

The major subroutine needed is *pseudo-remainder* computation. Recall that given $f(x), g(x)$ in $\mathbf{k}[x]$, the *pseudo-division with remainder* of f w.r.t. g computes $q, r \in \mathbf{k}[x]$ with

$$\mathsf{lc}(g)^{1+\mathsf{deg}(f)-\mathsf{deg}(g)} f = qg + r, \quad \mathsf{deg}(r) < \mathsf{deg}(g), \tag{6.14}$$

assuming $g \neq 0$. The polynomials q and r are uniquely determined by the Equation (6.14) and we also denote r by $\mathsf{prem}(f, g, x)$, which can be computed by the Algorithm 8. The polynomials q and r are called respectively the *pseudo-quotient* and *pseudo-remainder* of $f(x)$ w.r.t. $g(x)$.

Algorithm 8: Compute the pseudo-remainder $\mathsf{prem}(f, g, x)$

Input : polynomials $f, g \in \mathbf{k}[x]$ such that $\mathsf{deg}(f) \geq \mathsf{deg}(g) > 0$
Output : the pseudo-remainder of f by g in x

```
1 r ← f
2 for i ← deg(f) − deg(g) down to 0 do
3 |   r ← lc(g)r − lc(f)x^i g
4 return r
```

Algorithm 8 runs in $\mathsf{deg}(f) - \mathsf{deg}(g) + 1$ iterations. Iteration i costs $\mathsf{deg}(g) + i + 1$ multiplications to compute $\mathsf{lc}(g)\, r$; it costs $\mathsf{deg}(g) + 1$ multiplications to compute $\mathsf{lc}(f)x^i g$ and $\mathsf{deg}(g) + i + 1$ subtractions to compute r. Hence the total number of field operations performed to compute the pseudo-remainder $\mathsf{prem}(f, g, x)$ is

$$\begin{aligned}
&\sum_{i=0}^{\mathsf{deg}(f)-\mathsf{deg}(g)} (\mathsf{deg}(g) + i + 1) + (\mathsf{deg}(g) + 1) + (\mathsf{deg}(g) + i + 1) \\
&= \sum_{i=0}^{\mathsf{deg}(f)-\mathsf{deg}(g)} (3\,\mathsf{deg}(g) + 2i + 3) \\
&= (3\,\mathsf{deg}(g) + 3 + \mathsf{deg}(f) - \mathsf{deg}(g))(\mathsf{deg}(f) - \mathsf{deg}(g) + 1) \\
&= (\mathsf{deg}(f) + 2\,\mathsf{deg}(g) + 3)(\mathsf{deg}(f) - \mathsf{deg}(g) + 1). \qquad (6.15)
\end{aligned}$$

Thus, we obtain the following complexity result.

Lemma 21. *The pseudo-remainder of f, $g \in \mathbf{k}[x]$ with $\mathsf{deg}(f) \geq \mathsf{deg}(g)$ can be computed with $O(\mathsf{deg}(f)^2)$ arithmetic operations in $\mathbf{k}$.*

For $F, G \in \mathbf{k}[x]$ with $\mathsf{deg}(F) \geq \mathsf{deg}(G) > 0$, like the Euclidean algorithm, Brown's subresultant algorithm costs $3\mathsf{deg}(F)^2 + O(\mathsf{deg}(F))$ field operations in $\mathbf{k}$, as implied by [25]. Theorem 11 is a slightly refined cost estimate, but only for the case of finite fields, in which the sizes of coefficients are bounded. Let ℓ be the degree of G, and denote by d_i the degree of i-th subresultant $S_i = \mathsf{subres}_i(F, G, x)$ for $0 \leq i < \ell$. For convenience, we write $d_\ell = \ell = \mathsf{deg}(G)$ and $d_{\ell+1} = \mathsf{deg}(F)$.

Theorem 11. *Algorithm 7 uses*

$$d_{\ell+1}^2 + d_\ell^2 + d_{\ell+1} d_\ell + O(d_{\ell+1})$$

operations in $\mathbf{k}$, where $d_{\ell+1} = \mathsf{deg}(F) \geq d_\ell = \mathsf{deg}(G)$.

Proof. We first consider the *normal* case, where degree drops exactly by one at each iteration of Algorithm 7, that is, $d_i = i$ for $0 < i < \ell$. Note that in the normal case, Line 6 will get skipped since all subresultants are regular. Thus the total cost is

$$\begin{aligned}
&\sum_{i=1}^{\ell}(d_{i+1} + 2\,d_i + 3)(d_{i+1} - d_i + 1) \\
&= (d_{\ell+1} + 2\,d_\ell + 3)(d_{\ell+1} - d_\ell + 1) + \sum_{i=1}^{\ell-1}(d_{i+1} + 2\,d_i + 3)(d_{i+1} - d_i + 1) \\
&= (d_{\ell+1} + 2\,d_\ell + 3)(d_{\ell+1} - d_\ell + 1) + \sum_{i=1}^{\ell-1}(6\,i + 8) \\
&= (d_{\ell+1} + 2\,d_\ell + 3)(d_{\ell+1} - d_\ell + 1) + (3\,d_\ell^2 + 5\,d_\ell - 8) \\
&= d_{\ell+1}^2 + d_{\ell+1}d_\ell + d_\ell^2 + 4\,d_{\ell+1} + 4\,d_\ell - 5 \\
&\in d_{\ell+1}^2 + d_{\ell+1}d_\ell + d_\ell^2 + O(d_{\ell+1}).
\end{aligned}$$

Now we consider general cases. Assume that there exits a block of subresultants satisfying the following conditions (1) S_{k+1} is regular, (2) S_k has degree $e < k$. In this case, we have $S_{k-1} = \cdots = S_{e+1} = 0$ and S_e is regular. At Line 6, the algorithm computes $S_e = \alpha\, S_{k-1}$ with the cost at most

$$e + 1 + 2\log_2(k - e + 1),$$

where α is a power of a field element. [5] At Line 8, the algorithm computes $S_{e-1} = \beta\,\mathsf{prem}(S_{k+1}, S_k, x)$ with the cost

$$(k + 2\,e + 3)(k - e + 1) + 2\log_2(k - e + 1)$$

where β is a power of a field element. In this case, the total cost to compute S_{k-1}, …, S_e, S_{e-1} is

$$\begin{aligned}
&e + 1 + (k + 2\,e + 3)(k - e + 1) + 4\log_2(k - e + 1) \\
&= k^2 + ke - 2\,e^2 + 4\,k + 4 + 4\log_2(k - e + 1). \qquad (6.16)
\end{aligned}$$

However if subresultants S_i's are regular for all $e \le i \le k$, then the total cost to

[5] By repeated squaring, x^n can be computed in $2\log_2(n)$ multiplications.

compute S_{k-1}, ..., S_e, S_{e-1} would be

$$\begin{aligned}
&\sum_{i=e}^{k}(d_{i+1}+2\,d_i+3)(d_{i+1}-d_i+1)+(k-e+1)\\
&=\sum_{i=e}^{k}(6\,i+8)+(k-e+1)\\
&=3\,k^2+12\,k-3\,e^2-6\,e+9 \qquad (6.17)
\end{aligned}$$

Thus, according to Equation (6.17) and Equation (6.16), the cost difference for computing $S_{k-1},\ldots,S_{e-1}$ between the normal case and the general case is

$$\begin{aligned}
&3\,k^2+12\,k-3\,e^2-5\,e+9-(k^2+ke-2\,e^2+4\,k+4+4\log_2(k-e+1))\\
&=(2\,k-e)(k-e)+4\,k-2\,e+5+4(k-e-\log_2(k-e+1))>0
\end{aligned}$$

since $k>e\geq 0$ and $x\geq\log_2(1+x)$ for $x>0$.

Note that the subresultants with index smaller than $e-1$ will only depend on S_e and S_{e-1}. Hence one can proceed to the next block in the similar manner. In summary, the total cost for the normal case is strictly larger than the non-normal cases. Consequently, the Brown's subresultant algorithm runs in

$$d_{\ell+1}^2+d_{\ell+1}d_\ell+d_\ell^2+O(d_{\ell+1}),$$

field arithmetic operations. □

6.6 The complexity of FFT based subresultant chain construction

In this section, we analyze the memory consumption and the cost for constructing the subresultant chain of two dense multivariate polynomials P,Q in $\mathbb{Z}_p[x_1,\ldots,x_n,y]$ with the routines shown in Figure 6.1.

To simplify our presentation, assume that

(*a*) $\max(\mathsf{deg}(P,x_i),\mathsf{deg}(Q,x_i))\leq d_i$, for $1\leq i\leq n$,

(*b*) $\mathsf{deg}(P,y)=d_{n+2}\geq\mathsf{deg}(Q,y)=d_{n+1}>0$.

In practice, the FFT based subresultant chain of P and Q is stored in a data structure, called the *subresultant cube* of P and Q, in which a subresultant or one of its

coefficients can be interpolated via inverse FFTs whenever necessary. Quite often, the size of the cube becomes really large. Hence it is necessary to know if such an FFT based approach works.

For dense polynomials P and Q, the degree bounds given by the Sylvester bound (Equation (6.5)), the row bound (Equation (6.6)) and the column bound (Equation (6.7)) are in fact the same:

$$\mathsf{deg}(\mathsf{res}(P, Q, y), x_i) \leq (d_{n+2} + d_{n+1})d_i \quad \text{for } 1 \leq i \leq n. \tag{6.18}$$

Define $e_i = (d_{n+2} + d_{n+1})d_i + 1$ for $1 \leq i \leq n$. The Kronecker substitution δ with respect to $(e_1, \ldots, e_n)$ can be used to map P and Q to bivariate polynomials $F = \delta(P)$ and $G = \delta(Q)$ in $\mathbb{Z}_p[x, y]$. The following proposition claims that the subresultants of F and G in y can be recovered by the inverse of δ with respect to $(e_1, \ldots, e_n)$.

Proposition 20. *The following properties hold:*

(*a*) $\delta(\mathsf{subres}_j(P, Q, y)) = \mathsf{subres}_j(F, G, y)$, *for any* $0 \leq j < d_{n+1}$,

(*b*) $\mathsf{deg}(F, x) \leq d_1 + e_1 d_2 + \cdots + e_1 \ldots e_{n-1} d_n$,

(*c*) $\mathsf{deg}(G, x) \leq d_1 + e_1 d_2 + \cdots + e_1 \ldots e_{n-1} d_n$.

Proof. By the choice of e_i, we have

$$\mathsf{deg}(\mathsf{subres}_j(P, Q, y), x_i) \leq \mathsf{deg}(\mathsf{res}(P, Q, y), x_i) < e_i,$$

for any $0 \leq j < d_{n+1}$ and $1 \leq i \leq n$. According to Proposition 11, δ from $\mathbb{Z}_p[x_1, \ldots, x_n]$ to $\mathbb{Z}_p[x]$ satisfies properties

(1) $\delta(1) = 1$,

(2) $\delta(a + b) = \delta(a) + \delta(b)$,

(3) $\delta(ab) = \delta(a)\delta(b)$, whenever $\mathsf{deg}(ab, x_i) < e_i$ for all i.

With the spirit of Theorem 10 [6], we have $\delta(\mathsf{subres}_j(P, Q, y)) = \mathsf{subres}_j(F, G, y)$, which proves (*a*). Both (*b*) and (*c*) follow from the definition of δ, since $\delta(x_1^{d_1} \cdots x_n^{d_n}) = x^{d_1 + e_1 d_2 + \cdots + e_1 \ldots e_{n-1} d_n}$ holds. □

[6] By the definition, each coefficient of subresultants is a minor of the Sylvester matrix. These three properties grantee that δ is commutable with the operator of computing subresultants.

From the above proposition, we derive the degree bound B of $\mathsf{res}(F, G, y)$ as

$$\mathsf{deg}(\mathsf{res}(F, G, y), x) \leq B = (d_{n+1} + d_{n+2})(d_1 + \sum_{i=2}^{n} e_1 \cdots e_{i-1} d_i) \tag{6.19}$$

Let m be the smallest power of 2 such that $m > B$ and ω be an m-th primitive root of unity in $\mathbb{Z}_p$. Then the FFT size for evaluating F and G at x is m. Since translations on F and G do not change their partial degrees in each variable, we assume that no translation is needed for F and G. For each evaluation $x = \omega^i$, the size of the subresultant chain of $F(\omega^i, y)$ and $G(\omega^i, y)$ is $d_{n+1}(d_{n+1} + 1)/2$, with the completely dense encoding. From this, we derive the size of the evaluation cube:

Theorem 12. *Let P, Q in $\mathbb{Z}_p[x_1, \ldots, x_n, y]$ be dense multivariate polynomials such that $\mathsf{deg}(P, y) = d_{n+2} \geq \mathsf{deg}(Q, y) = d_{n+1}$ and $\max(\mathsf{deg}(P, x_i), \mathsf{deg}(Q, x_i)) \leq d_i$ for each $1 \leq i \leq n$. The size of the evaluation cube for computing the subresultant of P and Q in y (through Kronecker's substitution) is*

$$\frac{m d_{n+1}(1 + d_{n+1})}{2}, \tag{6.20}$$

where m is the smallest power of 2 such that

$$\begin{aligned} m &> (d_{n+1} + d_{n+2})(d_1 + \sum_{i=2}^{n} e_1 \cdots e_{i-1} d_i) \\ &= (d_{n+2} + d_{n+1}) \left(d_1 + \sum_{i=2}^{n} d_i \prod_{j=1}^{i-1} (d_{n+2} d_j + d_{n+1} d_j + 1) \right). \end{aligned}$$

When $d_{n+2} = d_{n+1} = \cdots = d_1 = d$, the FFT size is $\Theta(2^n d^{2n})$ and the size of the evaluation cube is $\Theta(2^n d^{2n+2})$.

In Table 6.1 we list the FFT degree required and the evaluation cube size for computing the subresultant chain of P and Q, when n can be regarded as the number of "parameters", d is the partial degree for each variable. Since the cube size is exponential with respect to the number n of parameters, the number of variables involved should be at most 6, (five of which are parameters).

Theorem 13. *Let P, Q in $\mathbb{Z}_p[x_1, \ldots, x_n, y]$ be dense multivariate polynomials such that $\mathsf{deg}(P, y) = d_{n+2} \geq \mathsf{deg}(Q, y) = d_{n+1}$ and $\max(\mathsf{deg}(P, x_i), \mathsf{deg}(Q, x_i)) \leq d_i$ for each $1 \leq i \leq n$. The number of field operations in $\mathbb{Z}_p$ for computing the subresultant*

n	d	FFT Degree	Cube Size	n	d	FFT Degree	Cube Size
1	80	14	203	3	6	19	42
1	100	15	632	3	8	22	576
1	120	15	908	3	10	23	1760
1	140	16	2468	4	5	23	480
2	15	18	120	4	6	25	2688
2	20	20	840	5	3	22	96
2	25	21	2600	5	4	26	2560

Table 6.1: The FFT degree required and the evaluation cube size in megabytes

of P and Q in y is

$$O(m \log m(d_{n+1} + d_{n+2} + 2) + m(d_{n+1}^2 + d_{n+2}^2 + d_{n+1}d_{n+2}))$$

where m is the smallest power of 2 such that

$$\begin{aligned} m &> (d_{n+1} + d_{n+2})(d_1 + \sum_{i=2}^{n} e_1 \cdots e_{i-1} d_i) \\ &= (d_{n+2} + d_{n+1}) \left(d_1 + \sum_{i=2}^{n} d_i \prod_{j=1}^{i-1} (d_{n+2} d_j + d_{n+1} d_j + 1) \right). \end{aligned}$$

Proof. We use the same notations as in Proposition 20, and define $\beta = d_1 + e_1 d_2 + \cdots + e_1 \ldots e_{n-1} d_n$. We observe that there is no cost (in terms of operations in $\mathbb{Z}_p$) for converting the polynomials P and Q into bivariate polynomials F and G.

According to Remark 6, the cost to apply a linear translation to F (resp. G) is $O(K_1)$ (resp. $O(K_2)$) with $K_1 := (d_{n+2} + 1)\beta \log \beta$ and $K_2 := (d_{n+1} + 1)\beta \log \beta$. If the number of valid translations in $\mathbb{Z}_p$ is at least $p/2$, then the expected number of trials is 2. This implies that the expected cost of applying linear translations to F and G is $O(K_1 + K_2)$.

The cost to evaluate F by means of univariate FFTs of size m is $O(K_3)$ with $K_3 = (d_{n+2} + 1)m \log m$. Similarly, the cost to evaluate G is $O(K_4)$, with $K_4 = (d_{n+1} + 1)m \log m$.

Finally, the cost to run the subresultant algorithm for each evaluation point is $O(K_5)$ with $K_5 = m(d_{n+2}^2 + d_{n+1}d_{n+2} + d_{n+1}^2 + O(d_{n+2}))$. It is clear that $K_1 + K_2$ is dominated by $K_3 + K_4$. Hence the total cost is

$$O(m \log m(d_{n+1} + d_{n+2} + 2) + m(d_{n+1}^2 + d_{n+2}^2 + d_{n+1}d_{n+2})).$$

□

When $d_{n+2} = d_{n+1} = \cdots = d_1 = d$, the cost to build the FFT based evaluation cube is $O(2^n d^{2n+2})$.

6.7 Implementation and experimental results

In this section, we present our CUDA implementation of constructing FFT based subresultant cube for bivariate polynomials.

The first step is to evaluate the input polynomials $F, G \in \mathbb{Z}_p[x, y]$. For our implementation, univariate polynomials are all dense, encoded as a vector of coefficients. Multivariate dense polynomials with n variables are encoded recursively as a univariate polynomial with $(n-1)$-variable polynomials as coefficients. For example, $F = 1 + 2x + 3xy + 4x^2 + 5y^2$ can encoded as a vector $[1, 2, 4, 0, 3, 0, 5, 0, 0]$, read as

$$F = (1 + 2x + 4x^2) + (0 + 3x + 0x^2)y + (5 + 0x + 0x^2)y^2.$$

We have realized a CUDA subroutine `list_fft_univariate` to perform a list of univariate FFTs on a list polynomials of the same size. For $n = 2^k$ and $q \geq 1$, the Stockham DFT factorization implies

$$I_q \otimes \mathrm{DFT}_n = \prod_{i=0}^{k-1} (I_q \otimes \mathrm{DFT}_2 \otimes I_{2^{k-1}})(I_q \otimes D_{2,2^{k-i-1}} \otimes I_{2^i})(I_q \otimes L_2^{2^{k-i}} \otimes I_{2^i}), \quad (6.21)$$

According to Section 5.3 and Section 5.5 of Chapter 5, `list_fft_univariate` can be implemented by extending the three CUDA kernels used in Section 5.5.

Example 24. *Let $F = a(x) + b(x)y + c(x)y^2 + d(x)y^3$ be a bivariate polynomial where a, b, c, d, have degree less than 8. Let ω be a 8-th primitive root of unity. After evaluating $F(x, y)$ at $x = (1, w, w^2, w^3, w^4, w^5, w^6, w^7)$, the layout of the evaluation data is*

$$M = \begin{bmatrix} a(1) & a(\omega) & a(\omega^2) & a(\omega^3) & a(\omega^4) & a(\omega^5) & a(\omega^6) & a(\omega^7) \\ b(1) & b(\omega) & b(\omega^2) & b(\omega^3) & b(\omega^4) & b(\omega^5) & b(\omega^6) & b(\omega^7) \\ c(1) & c(\omega) & c(\omega^2) & c(\omega^3) & c(\omega^4) & c(\omega^5) & c(\omega^6) & c(\omega^7) \\ d(1) & d(\omega) & d(\omega^2) & d(\omega^3) & d(\omega^4) & d(\omega^5) & d(\omega^6) & d(\omega^7) \end{bmatrix}$$

The i-th column corresponds the univariate polynomial $F(\omega^i, y)$ in y for $i = 0 \cdots 7$.

Transposing matrix M gives

$$M^t = \begin{bmatrix} a(1) & b(1) & c(1) & d(1) \\ a(\omega) & b(\omega) & c(\omega) & d(\omega) \\ a(\omega^2) & b(\omega^2) & c(\omega^2) & d(\omega^2) \\ a(\omega^3) & b(\omega^3) & c(\omega^3) & d(\omega^3) \\ a(\omega^4) & b(\omega^4) & c(\omega^4) & d(\omega^4) \\ a(\omega^5) & b(\omega^5) & c(\omega^5) & d(\omega^5) \\ a(\omega^6) & b(\omega^6) & c(\omega^6) & d(\omega^6) \\ a(\omega^7) & b(\omega^7) & c(\omega^7) & d(\omega^7) \end{bmatrix}.$$

We assume that the leading coefficient $d(x)$ in y of $F(x, y)$ does not vanish at any power of ω. Then each row of the matrix M^t can be seen as a univariate polynomial of degree 3.

As illustrated in the above example, the second step is to transpose the evaluated F and G for preparing the subresultant constructions. The major cost is to compute the subresultant chains at all evaluation points. To accelerate the overall performance, we implemented the Brown's subresultant algorithm in CUDA for computing a sequence of subresultant chains in a highly parallel manner and we present two different approaches for this task.

Coarse-grained approach The most direct way is to let each CUDA thread run a univariate Brown's subresultant algorithm. We called it the coarse-grained approach. This approach works all the time and for practical problems the number of threads can easily reach a big value, say 2^{15}, which can bring a significant speedup. However, there is a potential problem for this approach, due to unfavored memory access pattern to the GPU memory space, or more precisely threads in a thread warp are always accessing different memory regions.

Fine-grained approach The second approach is to break a list of univariate Brown's subresultant computations into a sequence of lists of univariate polynomial pseudo-divisions. Provided that at each step all A_j have the same degree and all B_j have the same degree [7], Algorithm 7 could be turned into the list version Algorithm 9. Initially, all F_j have the same degree according to the choice of ω, (as all G_j do).

For two univariate polynomials $P(x)$ and $Q(x)$, let S_{k_1}, ..., S_{k_ℓ} be all the regular resultants of P and G, such that $0 \leq k_1 < \cdots < k_\ell < \mathsf{deg}(Q)$. Then $(k_1, \ldots, k_\ell)$ is

[7] We distinguish 0 and nonzero constant polynomials. The degree of a nonzero constant is 0, while the degree of 0 is -1.

called the degree sequence of P and Q. In fact, they are the degrees of all remainders in the Euclidean algorithm with the input P and Q. Therefore, the above generic assumption says: the degree sequences of all images $F(\omega^i, y)$ and $G(\omega^j, y)$ are the same.

Algorithm 9: Running a list of Brown's algorithm in parallel

Input : A list of pairs of polynomials $F_j, G_j \in \mathbf{k}[x]$ for $0 \le j < m$ such that $\mathsf{deg}(F_j) \ge \mathsf{deg}(G_j) > 0$, all F_j having the same degree, and all G_j having the same degree too.

Output : the subresultant chain of F_j and G_j, for $0 \le j < m$.

1 $S_i^j \leftarrow 0$ for $0 \le i < \mathsf{deg}(G_j)$ and $0 \le j < m$
2 $B_j \leftarrow \mathsf{prem}(F_j, -G_j, x)$, $A_j \leftarrow G_j$, $\alpha \leftarrow \mathsf{deg}(F_j) - \mathsf{deg}(G_j)$, for $0 \le j < m$
3 **while** $B_j \neq 0$ **do**
4 $\quad d \leftarrow \mathsf{deg}(A_j)$, $e \leftarrow \mathsf{deg}(B_j)$, $\delta \leftarrow d - e$, for $0 \le j < m$
5 $\quad S_{d-1}^j \leftarrow B_j$ for $0 \le j < m$
6 $\quad S_e^j \leftarrow \mathsf{lc}(A_j)^{\alpha(1-\delta)} \mathsf{lc}(B_j)^{\delta-1} B_j$ for $0 \le j < m$
7 $\quad$ **if** $e = 0$ **then break**
8 $\quad B_j \leftarrow \mathsf{lc}(A_j)^{-\alpha\delta-1} \mathsf{prem}(A_j, -B_j, x)$, $A_j \leftarrow S_e^j$, $\alpha \leftarrow 1$, for $0 \le j < m$
9 **return** S_i^j *for* $0 \le i < \mathsf{deg}(G_j)$, *for* $0 \le j < m$

Note that in Algorithm 9, the generic assumption implies

Line 3. If $B_j = 0$ for some $0 \le j < m$, then all B_j are zero.

Line 4. All A_j have the same degree d, and all B_j have the same degree e.

Line 6. All S_e^j can be computed by a CUDA kernel.

Line 8. All pseudo-divisions can be computed by a CUDA kernel.

The key subroutine is to perform a list of pseudo-divisions in a fine-grained way.

Example 25. *Let* $f = a_3x^3 + a_2x^2 + a_1x + a_0$ *and* $g = b_2x^2 + b_1x + b_0$. *To obtain the pseudo-remainder* $\mathsf{prem}(f, -g, x)$ *of* f *and* g, *we compute*

(1) $h_2 = -b_2 f + a_3 x g = c_2x^2 + c_1x + c_0$,

(2) $h_1 = -b_2h_2 + c_2g = d_1x + b_0$.

Alternatively, the pseudo-remainder $\mathsf{prem}(f, -g, x)$ *can be computed in two steps:*

$$(S_1) \quad c_2 = \begin{vmatrix} a_3 & a_2 \\ b_2 & b_1 \end{vmatrix}, \quad c_1 = \begin{vmatrix} a_3 & a_1 \\ b_2 & 0 \end{vmatrix}, \quad c_0 = \begin{vmatrix} a_3 & a_0 \\ b_2 & 0 \end{vmatrix};$$

$$(S_2) \quad d_1 = \begin{vmatrix} c_2 & c_1 \\ b_2 & b_1 \end{vmatrix}, \quad d_0 = \begin{vmatrix} c_2 & c_0 \\ b_2 & b_0 \end{vmatrix}.$$

As illustrated in the above example, the basic unit is to perform a single reduction step. The following CUDA kernel `list_reduce_kernel` takes two lists of univariate polynomials LF and LG, and computes

$$h_i = \mathsf{lc}(g_i) f_i - \mathsf{lc}(f_i) x^{\mathsf{deg}(f_i) - \mathsf{deg}(g_i)} g_i,$$

where f_i is the i-th polynomial in LF and g_i is the i-th polynomial in LG. We assume that polynomials in LG have the same degree dG, and polynomials in LF have the same degree dF. The result LH consists of h_i's computed.

```
__global__ void
list_reduce_kernel(int *LH, int dF, int *LF, int dG, int *LG, int p)
{
    int bid = blockIdx.x;                       // block index
    int tid = bid * blockDim.x + threadIdx.x; // thread index
    int qtid = tid / dF;                        // pair index
    int rtid = tid % dF;

    int *F = LF + qtid * (dF + 1);              // first polynomial
    int *G = LG + qtid * (dG + 1);              // second polynomial
    int *H = LH + qtid * dF;                    // output polynomial

    // The configuration is the following
    //         u ..... a
    //   v ..... b
    // where a is the current coefficient to be eliminated,
    // b is the current leading coefficient (nonzero),
    // and u, v are cofficients to be adjusted.
    // For each pair (u, v), compute a * v - u * b mod p
    // and store it to H[rtid];
```

```
    int dgap = dF - dG;
    int a = F[dF];                  // the leading coefficient of F
    int b = G[dG];                  // the leading coefficient of G
    int u = F[rtid];
    int v = ((rtid >= dgap) ? G[rtid - dgap] : 0);

    int t1 = mul_mod(a, v, p);              // t1 = a * v mod p
    int t2 = mul_mod(b, u, p);              // t2 = b * u mod p
    H[rtid] = sub_mod(t1, t2, p);
}
```

In terms of CUDA kernel `list_reduce_kernel`, a list of pseudo-remainders can be computed as in Algorithm 10 using a double-buffer method.

Algorithm 10: Computing a list of pseudo-remainders in parallel

Input : Two lists LF, LG of polynomials such that $LF[i] = F_i$, $LG[i] = G_i$ for $0 \leq i < m$, $\mathsf{deg}(F_i) \geq \mathsf{deg}(G_i) > 0$, all F_i having the same degree dF, and all G_i having the same degree dG.

Output : Compute the list LH of polynomials such that $LH[i] = H_i = \mathsf{prem}(F_i, -G_i, x)$ for $0 \leq i < m$.

```
// We use the double-buffer method, and the buffers
// LX and LY are of the same size as LF

// reduce once and store the result into LX
1 list_reduce(m, LX, dF, LF, dG, LG, p)
2 for d ← dF − 1 downto dG do
      // reduce once and store the result into LY
3     list_reduce(m, LY, d, LX, dG, LG, p)
      // switch the role of LX and LY
4     swap(LX, LY)
5 Copy out the result from LX to LH
```

In Figure 6.2, we report our preliminary experimentation on computing the subresultant chains of random dense square bivariate polynomials with partial degree d in both x and y. The characteristic of the finite field is $p = 943718401 = 225 \times 2^{22} + 1$. The second column shows the estimated size of the subresultant chain data structure in megabytes. The third column reports the time spent by the FFT based serial code in the C library `modpn`. In the fourth column, we list the timing of coarse-grained

d	$size$(MB)	$modpn$	$coarse$	$\frac{modpn}{coarse}$	$fine$	$\frac{modpn}{fine}$
10	0.0	0.000	0.030	0.0	0.010	0.0
15	0.2	0.010	0.040	0.2	0.010	1.0
20	0.7	0.040	0.030	1.3	0.010	4.0
25	2.3	0.110	0.030	3.6	0.020	5.5
30	3.4	0.120	0.030	4.0	0.020	6.0
35	9.3	0.310	0.060	5.1	0.040	7.7
40	12.2	0.400	0.060	6.6	0.040	10.0
45	15.5	0.470	0.070	6.7	0.050	9.3
50	38.3	1.330	0.160	8.3	0.060	22.1
55	46.4	1.300	0.170	7.6	0.080	16.2
60	55.3	1.900	0.210	9.0	0.100	18.9
65	130.0	3.410	0.380	8.9	0.140	24.3
70	150.9	3.810	0.530	7.1	0.170	22.4
75	173.4	4.290	0.590	7.2	0.190	22.5
80	197.5	4.730	0.670	7.0	0.210	22.5
85	223.1	5.500	0.550	10.0	0.230	23.9
90	250.3	7.460	0.840	8.8	0.260	28.6
95	558.1	12.710	1.840	6.9	0.500	25.4
100	618.8	13.790	2.120	6.5	0.540	25.5
105	682.5	15.650	2.280	6.8	0.590	26.5
110	749.4	16.280	2.480	6.5	0.650	25.0
115	819.4	17.520	2.750	6.3	0.690	25.3

Figure 6.2: Bivariate subresultant chain construction in seconds

implementation, and its speedup factor to `modpn` is listed in the fifth column. The sixth column reports the timing of fine-grained implementation and its speedup factor to `modpn` is listed in the last column.

As shown in Figure 6.2, when the partial degree d is larger than 20, GPU code starts to outperform the C code. The coarse-grained GPU code has a limited speedup factor, due to the lack of parallelism. Notice that the speedup factor starts to drop as d grows from $d = 90$. The main reason is that threads inside each thread block are constructing the subresultant chain for different evaluation images, which implies much more accesses to the global memory space. On the other hand, the fine-grained GPU code achieves a speedup factor approximately 20 to 28 when $d \geq 50$ and the speedup factor also maintains for larger partial degrees.

The reason that fine-grained approach outperforms the coarse-grained one is in the fine-grained implementation threads in a thread block are cooperatively handling the same task, i.e. mainly running `list_reduce_kernel`. Figure 6.3 shows the statistics for constructing bivariate subresultant chain data structure with $d = 100$.

The hot spot of the implementation is the CUDA kernel `list_reduce_kernel` (a.k.a. `list_poly_reduce_defective_ker`).

As discussed in Section 6.1, the other applicable approach for modular subresultant chain constructions is to map the multivariate problems into the trivariate ones. The evaluation step is then performed by means of bivariate FFTs. The main advantage is that this could use primitive roots of unity in lower order, which in turn enlarges the set of available Fourier prime numbers.

Given trivariate polynomials $F(x, y, z)$ and $G(x, y, z)$ in $\mathbf{k}[x, y, z]$, we first evaluate F and G at a valid grid for F and G,

$$\Theta = \{(\omega_1^i, \omega_2^j) \mid 0 \leq i < m_1, 0 \leq j < m_2\},$$

where ω_i is an m_i-th primitive root of unity, m_1 and m_2 are computed from the degree bound in x and y respectively. With the tool introduced in Chapter 5, the two dimensional $\mathrm{DFT}_{m_1 \times m_2}$ is defined as the tensor product of two one dimensional DFTs, and the row-column algorithm [26] can be expressed in terms of the tensor product as follows

$$\mathrm{DFT}_{m_1 \times m_2} = \mathrm{DFT}_{m_1} \otimes \mathrm{DFT}_{m_2} = (\mathrm{DFT}_{m_1} \otimes I_{m_2})(I_{m_1} \otimes \mathrm{DFT}_{m_2}).$$

Notice that Equation (6.21) is a general form of $I_{m_1} \otimes \mathrm{DFT}_{m_2}$ and

$$\mathrm{DFT}_{m_1} \otimes I_{m_2} = \prod_{i=0}^{k-1} (\mathrm{DFT}_2 \otimes I_{2^{k-1+s}}) \otimes (D_{2,2^{k-i-1}} \otimes I_{2^{i+s}}) \otimes (L_2^{2^{k-i}} \otimes I_{2^{i+s}}),$$

where $m_1 = 2^k$ and $m_2 = 2^s$. Thus, bivariate FFTs could be implemented using the CUDA kernels developed to compose a list of univariate FFTs.

After evaluating F and G at the grid Θ, we have $m_1 \times m_2$ pairs of univariate polynomials, for each of which we construct their subresultant chain by means of Brown's algorithm. Once again, we could do it in a coarse-grained manner or in a fine-grained one. In practice, the latter is tried first, and if an error is raised, then we switch the former.

In Figure 6.4, we report our experimentation on computing the subresultant chains of random dense square trivariate polynomials with partial degree d in x, y, and z. The characteristic of the finite field is $p = 943718401 = 225 \times 2^{22} + 1$. The second column shows the estimated size of the subresultant chain data structure in megabytes. The third column reports the time spent by the FFT based serial code in the C library

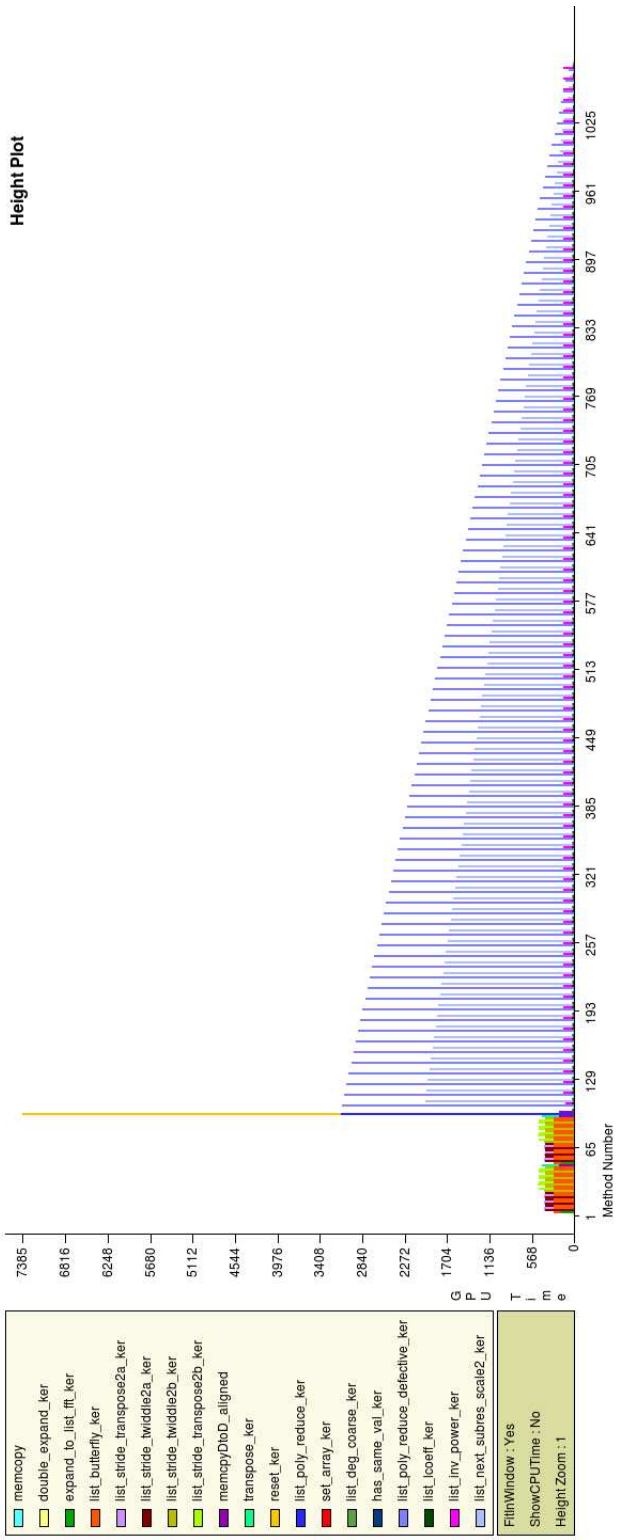

Figure 6.3: Fine-grained subresultant chain construction

n	$Size$(MB)	$modpn$	$coarse$	$\frac{modpn}{coarse}$	$fine$	$\frac{modpn}{coarse}$
6	0.2	0.020	0.040	0.5	0.010	2.0
7	1.3	0.160	0.040	4.0	0.010	16.0
8	1.8	0.160	0.040	4.0	0.010	16.0
9	9.0	0.860	0.050	17.2	0.030	28.6
10	11.2	0.850	0.060	14.1	0.020	42.5
11	13.8	0.940	0.070	13.4	0.030	31.3
12	16.5	1.070	0.070	15.2	0.050	21.4
13	78.0	6.170	0.190	32.4	0.120	51.4
14	91.0	5.380	0.230	23.3	0.130	41.3
15	105.0	6.290	0.250	25.1	0.150	41.9
16	120.0	7.070	0.290	24.3	0.160	44.1
17	544.0	32.090	1.170	27.4	0.670	47.8
18	612.0	34.670	1.290	26.8	0.730	47.4
19	684.0	36.980	1.250	29.5	0.770	48.0

Figure 6.4: Trivariate subresultant chain constructions in seconds

`modpn`. In the fourth column, we list the timing of coarse-grained implementation, and its speedup factor to `modpn` is listed in the fifth column. The sixth column reports the timing of fine-grained implementation and its speedup factor to `modpn` is listed in the last column.

From Figure 6.4, first we observe that the size of the subresultant chain grows rapidly with the partial degree d, as predicted in Section 6.6. In trivariate case, the `modpn` C implementation perform worse than in bivariate case. The coarse-grained implementation is approximately 25 faster, when the partial degree n is at least 13. The fine-grained implementation is approximately 47 times faster than the `modpn` C implementation. The underline reason is probably due to the complicated structure of the multivariate FFT and frequent data transpositions therein. This also suggests that it is worth to map large multivariate problems into bivariate or trivariate ones at the cost of Kronecker's substitutions.

6.8 Summary

In this chapter, we have proposed a complete modular algorithm to construct the subresultant chain of two multivariate polynomials over a finite field. This algorithm relies on Fast Fourier Transforms based evaluations and interpolations, and efficient implementation of Brown's subresultant algorithm.

We analyzed the various issues in this algorithm. Transforming multivariate poly-

d	MAPLE	`modpn`	`modpn` with GPU
20	4.080	0.112	0.040
25	12.509	0.240	0.036
30	32.566	0.204	0.060
35	74.480	0.496	0.100
40	160.246	0.808	0.120
45	315.996	0.944	0.156
50	594.525	1.620	0.244
55	-	2.461	0.276
60	-	2.732	0.396
65	-	5.640	0.596
70	-	6.341	0.700
75	-	7.140	0.916
80	-	8.001	1.052
85	-	7.380	1.376
90	-	9.113	1.664
95	-	16.529	2.408
100	-	18.213	2.636

Figure 6.5: Computing resultants of bivariate dense polynomials in seconds

nomials into univariate or bivariate ones can be accomplished by means of Kronecker's substitutions. Finding valid primitive roots of unity can be done via linear translations. For most practical problems, valid translations can be found when the characteristic of the finite field is sufficiently large.

Based on univariate and bivariate FFTs developed in Chapter 5, we have realized a coarse-grained Brown's subresultant algorithm and a fine-grained one. Our fine-grained implementation is approximately 25 times faster than `modpn` C counterpart for the bivariate inputs, and approximately 47 times faster for the trivariate inputs.

There are a number of problems to address from both the theoretical and implementation point of view. For example, it is of great interest to know how to predict if the fine-grained code will fail due to the break of the generic assumption proposed in Section 6.7. We observe that if we write the Fourier prime p as $p = c\,2^n + 1$, the generic assumption rarely breaks for random dense bivariate inputs, whenever c is large, such as a few hundreds.

Our initial plan was to construct subresultant chain cube via GPU kernels, then interpolate subresultants via inverse FFTs of `modpn`. In Figure 6.5, we compared the timing for computing the resultant of two random dense square polynomials with MAPLE code (Column 2), `modpn` serial code (Column 3), and `modpn` code with the subresultant chain cube constructed inside the GPU (Column 4). Comparing to

d	t_0	t_1	t_1/t_0
30	0.23	0.29	1.3
40	0.23	0.43	1.9
50	0.27	1.14	4.2
60	0.27	1.53	5.7
70	0.31	3.95	12.7
80	0.32	4.88	15.3
90	0.35	5.95	17.0
100	0.50	19.10	38.2
110	0.53	17.89	33.8
120	0.58	19.72	34.0

Figure 6.6: Computing resultants for bivariate dense polynomials in seconds

Figure 6.2, the speedup factor drops from 25 to 7. The major reason is that inside `modpn` all subresultant cubes have the same data layout while the cubes constructed inside GPU are different. There is an expensive conversion in between. A possible solution to this problem is to refactor the interface, and let the interpolation process reside in GPU too.

Further, observe that for the regular GCD algorithm developed in Chapter 3 not all subresultants need to be interpolated. It is possible to keep the subresultant chain cube inside the GPU global memory all the time and we interpolate subresultants or coefficients of their subresultants whenever needed.

6.9 Complementary Experimental Results

In this section, we include our recent experimental results in [66] for computing subresultant chain cubes and resultants. We use a GPU card Nvidia Telsa C2050, which is different from the one used in other sections, namely the Nvidia Geforce GTX 285. The other improvement is that we interpolate resultants by means of GPU based inverse FFT, which avoid transfering subresultant chain cubes back to the main memory.

Figure 6.6 reports the timing for computing resultant $\mathsf{res}(F_1, F_2, y)$ with bivariate random dense polynomials $F_1, F_2 \in \mathbb{Z}_p[x, y]$ such that $p = 469762049$ and $d = \mathsf{deg}(F_i, x) = \mathsf{deg}(F_i, y)$ for $i = 1, 2$. In the figure, the first column, labelled by d, shows the partial degree d. The second one, labelled by t_0, is the timing for GPU FFT-based scube method, which includes the time for moving result back to the main memory. The third column, labelled by t_1, shows the CPU FFT based scube serial C code in the `modpn` library [51]. The last column reports the ratio between

d	t_0	t_1	t_1/t_0
7	0.22	0.16	0.7
8	0.23	0.76	3.3
9	0.24	0.85	3.5
10	0.25	0.98	3.9
11	0.24	1.10	4.6
12	0.30	4.96	16.5
13	0.31	5.52	17.8
14	0.32	6.07	19.0
15	0.78	8.95	11.5
16	0.65	31.65	48.7
17	0.66	34.55	52.3
18	3.46	47.54	13.7
19	0.73	51.04	69.9
20	0.75	43.12	57.5

Figure 6.7: Computing resultants for trivariate dense polynomials in seconds

the two implementations. Note that all the resultants in this experimentation are computed with the fine-grained method and that we interpolate the resultants inside the GPU inverse FFT, keeping the subresultant chain cube inside the GPU global memory. The maximal speedup we achieve is approximately 38.

Figure 6.7 lists our experimental results for computing resultants for trivariate random dense polynomials in $\mathbf{k}[x, y, z]$. The first column shows the common partial degree d in x, y and z. The other three column have the same meaning as in the bivariate case. Note that all but $d = 15$ and $d = 18$ are based on fine-grained scube's. When the coarse-grained method is forced to be used, the speedup it achieves drops significantly.

We observe that the GPU based implementation achieves a much larger speedup factor in the trivariate case (approximately 70 for the best cases) than in the bivariate case (approximately 38). The underline reason may be that the GPU based implementation could take advantage of the assumption that the input are trivariate to avoid unnecessary data transpositions, in both FFT evaluations and the Brown's algorithm.

Chapter 7

Conclusions and Future Work

The computation of polynomial GCDs is at the core of the theory of regular chains and its application to polynomial system solving by means of triangular decomposition. In this context, polynomials take their coefficients in rings which have less algebraic structure than in the classical setting of unique factorization domains (UFDs).

The formalization of polynomial GCDs modulo (saturated ideals of) regular chains started less than twenty years ago, with the PhD thesis of Michael Kalkbrener. Our work relies on the definition proposed by Marc Moreno Maza, which is more suitable for algorithm design.

In this thesis, we have presented the first algorithm, called RGSZR, which, for the purpose of computing the so-called *regular GCDs*, is both practically efficient and based on fast polynomial arithmetic and modular techniques. We have reported on two implementations: a serial one in C language and parallel one supported by graphics card (GPU) code.

The latter one currently performs the main computational step of our algorithm, namely the construction of the so-called *subresultant cube*. This already yields promising experimental results. We are now working on completing this implementation such that the whole RGSZR can be supported by GPU code. From there, the natural question is how much a complete polynomial system solver (such as the `Triangularize` command of the `RegularChains` library) can take advantage of GPU support.

We have also estimated the algebraic complexity of the RGSZR algorithm, under standard genericity assumptions. In this setting and for the problem sizes that are of practical interest, our complexity estimate brings new favorable results. Relaxing these genericity assumptions (in particular the dimension zero assumption) is, of course, part of our future objectives.

Regular GCDs are used directly or indirectly in all main subroutines of the

`Triangularize` command. The design of the underlying Triade algorithm was motivated by controlling intermediate expression swell, in particular, by avoiding redundant computational branches. By generalizing the classical notion of polynomial primitivity from UFDs to general commutative rings (with unity) we have obtained new criteria for detecting that the saturated ideal of a regular chain is contained in another such saturated ideal. Although these criteria do not cover all possible cases, they have provided significant improvements in practice. Deciding saturated ideal inclusion (without computing generator systems as this can be extremely expensive) remains an open and very exciting question.

Appendix A

A Review of Concepts on Polynomial System Solving

In this chapter, we list some definitions and basic constructions appeared in the thesis.

A.1 Polynomials

A field $\mathbf{k}$ is an algebraic structure with notions of addition, subtraction, multiplication, and division, satisfying certain axioms. The multivariate polynomial ring $\mathbf{k}[x_1, \ldots, x_n]$ is formed from the set of polynomials in x_1, ..., x_n with coefficients in $\mathbf{k}$. A field $\mathbf{K}$ is said to be algebraically closed if every polynomial of degree one in $\mathbf{K}[x]$ has a root in $\mathbf{K}$. For example, the field of complex numbers $\mathbb{C}$ is algebraically closed, while the field of rational numbers $\mathbb{Q}$ is not.

A polynomial ideal $\mathcal{I}$ is a subset of $\mathbf{k}[x_1, \ldots, x_n]$ satisfying $fg \in \mathcal{I}$ for any $f \in \mathbf{k}[x_1, \ldots, x_n]$ and $g \in \mathcal{I}$. For any subset $G \subseteq \mathbf{k}[x_1, \ldots, x_n]$, the ideal generated by G, denoted by $\langle G \rangle$, is the set

$$\langle G \rangle = \{f \in \mathbf{k}[x_1, \ldots, \boldsymbol{x}_n] \mid f = \sum_{i=1}^{k} a_i g_i,\ a_i \in \mathbf{k}[x_1, \ldots, x_n], g_i \in G, \text{ and } k \geq 1\}.$$

When $G = \{g_1, \ldots, g_s\}$ is a finite set, we also write $\langle G \rangle = \langle g_1, \ldots, g_s \rangle$. The radical $\sqrt{\mathcal{I}}$ of an ideal $\mathcal{I}$ is defined as the set of polynomials

$$\{f \in \mathbf{k}[x_1, \ldots, x_n] \mid f^m \in \mathcal{I} \text{ for some } m \geq 1\},$$

which is also an ideal in $\mathbf{k}[x_1, \ldots, x_n]$.

Let $\mathbf{K}$ be algebraically closed and $\mathbf{k} \subseteq \mathbf{K}$. The algebraic variety or algebraic set of G, denoted by $V(G)$, is defined as

$$V(G) = \{(a_1, \ldots, a_n) \in \mathbf{K}^n \mid g(a_1, \ldots, a_n) = 0, \text{ for all } g \in G\},$$

which is the common zeros over $\mathbf{K}$ of polynomials in G. It is not hard to show $V(G) = V(\langle G \rangle) = V(\sqrt{\langle G \rangle})$.

We conclude this section by the celebrated Hilbert's Basis Theorem and Hilbert's Nullstellensatz.

Theorem 14 (Hilbert's Basis Theorem)**.** *Every ideal in* $\mathbf{k}[x_1, \ldots, x_n]$ *can be generated by a finite number of polynomials.*

Theorem 15 (Hilbert's Nullstellensatz)**.** *The ideal* $\mathcal{I} \subseteq \mathbf{k}[x_1, \ldots, x_n]$ *contains* 1 *if and only if the polynomials in* $\mathcal{I}$ *do not have any common zeros in* $\mathbf{K}^n$*, i.e.* $V(I) = \emptyset$*.*

The Nullstellensatz is a generalization of the fundamental theorem of algebra. The Basis Theorem implies that every algebraic set over field can be described as the set of common roots of finitely many polynomial equations. Hilbert's proof only shows the existence and does not give an algorithm to produce the finitely many basis polynomials. One can determine basis polynomials using the method of Gröbner bases.

A.2 Gröbner Basis

A Gröbner basis is a particular kind of generating subset of an ideal $\mathcal{I}$ in $\mathbf{k}[x_1, \ldots, x_n]$, which generalizes the Gaussian elimination for linear systems. The Gröbner basis theory for polynomial rings was developed by Bruno Buchberger in 1965, who named them after his advisor Wolfgang Gröbner.

Given the variable ordering $x_1 < \ldots < x_n$, let $M = \{x_1^{e_1} \cdots x_n^{e_n} \mid e_j \geq 0\}$ be the set of monomials generated by $x_1, \ldots, x_n$. A monomial order on M is a total order, satisfying the following two properties:

- If $u < v$ and w is any other monomial, then $uw < vw$.
- Every non-empty subset of M has a minimal element.

For example, the lexicographical order is defined as

$$x_1^{e_1} \cdots x_n^{e_n} <_{lex} x_1^{f_1} \cdots x_n^{f_n} \iff f_k > e_k \text{ and } e_i = f_i \text{ for } i = 1 \cdots k-1.$$

Given a polynomial f and a monomial ordering σ, the leading term of f, denoted by $\mathrm{LM}_\sigma(f)$, is the greatest monomial appearing in f. For an ideal $\mathcal{I} \subseteq \mathbf{k}[x_1, \ldots, x_n]$, a finite set $G \subset \mathcal{I}$ is a Gröbner basis of $\mathcal{I}$ w.r.t monomial ordering σ if the following equality holds

$$\langle \mathrm{LM}_\sigma(\mathcal{I}) \rangle = \langle \mathrm{LM}_\sigma(G) \rangle,$$

that is, the ideal generated by the leading monomials of polynomials in $\mathcal{I}$ equals the ideal generated by the leading monomials of polynomials in G.

The well-known Buchberger's algorithm is a method transforming a given set of generators for a polynomial ideal into a Gröbner basis with respect to some monomial order. The following list of questions can be answered by means of Gröbner basis computations.

Theorem 16. *Let $\mathcal{I}$ be a polynomial ideal in $\mathbf{k}[x_1, \ldots, x_n]$. Then the following problems are solvable by means of Gröbner basis computations.*

- *(Ideal Membership) Decide if $f \in \mathcal{I}$ for any $f \in \mathbf{k}[x_1, \ldots, x_n]$.*
- *(Radical Membership) Decide if $f \in \sqrt{\mathcal{I}}$ for any $f \in \mathbf{k}[x_1, \ldots, x_n]$.*
- *(Elimination) Compute a set of generators for the ideal $\mathcal{I} \cap \mathbf{k}[x_1, \ldots, x_i]$ for some $1 \leq i \leq n$.*

We emphasize that the above three problems can be extended or generalized to solve numerous problems in computational commutative algebra, invariant theory, etc.

Appendix B

GPU Programming with CUDA

In this chapter, we briefly introduce the technology of conducting general-purpose computations on graphics processing units (GPUs), with a focus on Nvidia CUDA (an acronym for Compute Unified Device Architecture) enabled graphics cards.

GPUs are high-performance many-core processors capable of very high computation and data throughput. They are specially designed for computer graphics applications and are well-known for their difficulty to program. Today's GPUs are general-purpose parallel processors with support for accessible programming interfaces and industry-standard languages such as C. Developers who port their applications to GPUs often achieve speedups of orders of magnitude compared to optimized CPU implementations.

In 2006, Nvidia Corporation released the initial version of CUDA, a parallel computing architecture. CUDA is the computing engine in Nvidia GPUs and is accessible to software developers through variants of standard C programming language. As of 2010, CUDA has been applied to broad-ranging applications including image and video processing, computational biology and chemistry, fluid dynamics simulation, ray tracing, etc.

This chapter is organized as follows. In Section B.1 and B.3, we present the programming and memory model of CUDA. Section B.2 introduces the CUDA hardware architecture. In the end, we introduce several key factors related to the performance of CUDA programs. Detailed explanations on CUDA can be accessed through the CUDA Programming Guide [3].

B.1 CUDA programming model

CUDA is a heterogeneous serial-parallel programming model. A CUDA program executes serial code on the host (CPU) interleaved with parallel threads execution on the device (GPU). A single CUDA kernel, or simply kernel, consists of a number (usually large) of threads, which run the same code. Each thread has a unique ID which is used to compute memory addresses and to make control decisions.

Example 26. *The following simple CUDA program increments each entry of a vector. It is a C program with some additional keywords. For instance, a kernel is defined using the declaration specifier* __global__.

```
// kernel definition
__global__ void increment_dev(int *X_d, int n)
{
  int idx = threadIdx.x;
  X_d[idx] += 1;          // increment the entry idx
}
```

Each of the threads that executes this kernel is given a unique thread ID which is accessible within the kernel through the built-in variable `threadIdx`. *When called, the above kernel is executed in parallel by n different CUDA threads.*

```
void increment_host(int *X, int n)
{
  // first copy data from host X to device X_d,
  // to be completed later

  // kernel invocation
  increment_dev<<<1, n>>>(X_d, n);

  // copy data back from devicen X_d back to host X,
  // to be completed later
}
```

To manage a large number of threads that can work cooperatively, CUDA groups them into *thread blocks*. Each block contains up to 512 threads that can share data in fast on-chip memory and synchronize with barriers, as explained in Figure B.1. For convenience, `threadIdx` is a 3-component vector, so that threads can be identified

using a one-dimensional, two-dimensional, or three-dimensional thread index. This provides a natural way to invoke computation across the elements of a data structure, such as a vector or a matrix.

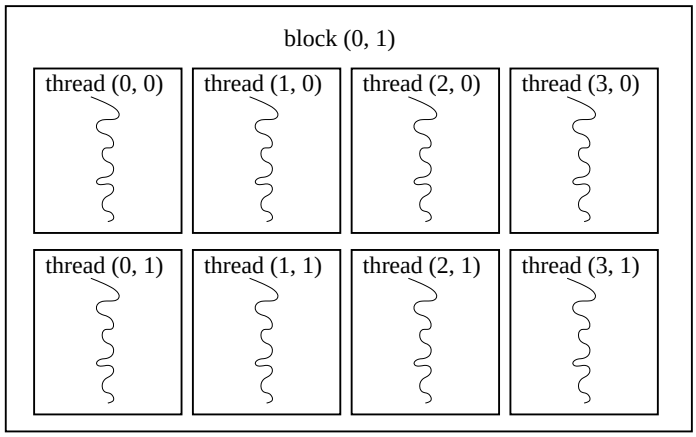

Figure B.1: A block is composed of threads

These multiple blocks are organized into a one-dimensional or two-dimensional grid of thread blocks as illustrated by Figure B.2. The dimension of the grid is specified by the first parameter of the $<<< \cdots >>>$ syntax. Each block within the grid can be identified by a one-dimensional or two-dimensional index accessible within the kernel through the built-in variable `blockIdx`. The dimension of the thread block is accessible within the kernel through the built-in variable `blockDim`.

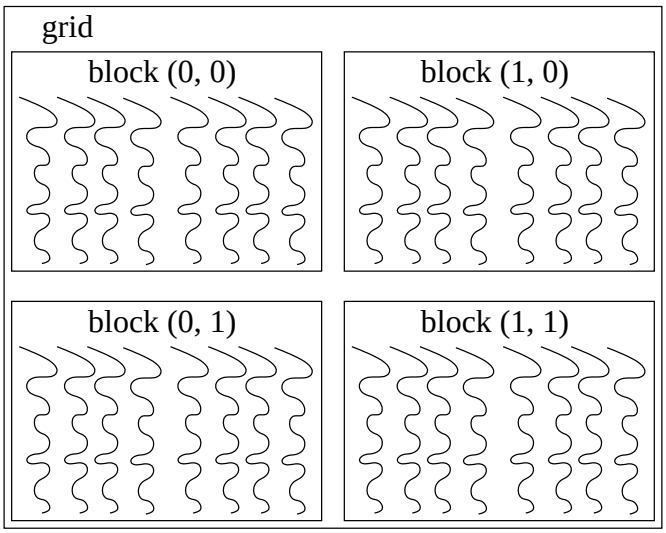

Figure B.2: A grid is composed of blocks

B.2 CUDA architecture

The CUDA architecture is built around a scalable array of multithreaded multiprocessors, as illustrated in Figure B.3. A multiprocessor consists of eight scalar processor (SP) cores, or arithmetic logic unit (ALU) as shown in Figure B.4. The multiprocessor creates, manages, and executes concurrent threads in hardware with zero scheduling overhead. The multiprocessor employs an SIMT (single-instruction multiple-thread) architecture: the multiprocessor maps each thread to one scalar processor core, and each thread executes the same program independently on different data.

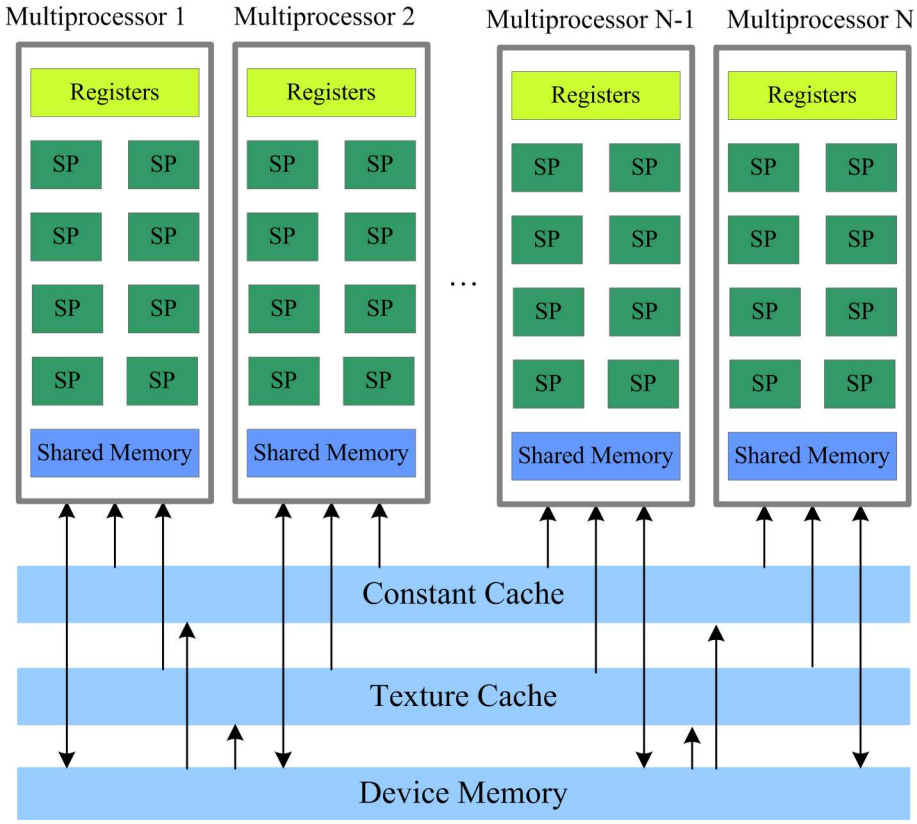

Figure B.3: Nvidia G80 architecture

When a CUDA program on the host CPU invokes a CUDA kernel, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one (and only one) multiprocessor. As thread blocks terminate, new blocks are launched on the vacant multiprocessors.

Since thread blocks are arranged in a grid and execute independently from each other, an important consequence is that there is no synchronization among the threads

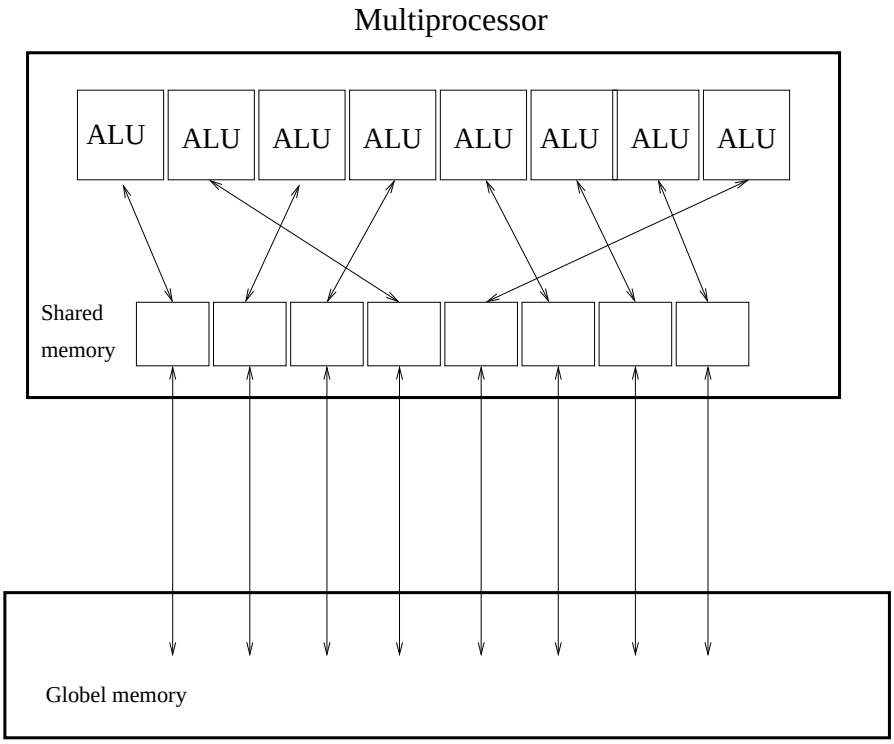

Figure B.4: Multiprocessor inside a Nvidia GPU

from different thread blocks in a grid. Therefore, algorithms with sequentially dependent computations must be decomposed into a series of CUDA kernels. Meanwhile, threads in the same block of a kernel are allowed to cooperate using barrier synchronization, with the single instruction __syncthreads. This paradigm enables transparent scalability for a kernel since all thread blocks can be scheduled to any of the available multiprocessors. This scheduling is determined by the runtime system.

B.3 CUDA memory model

CUDA memory hierarchy is mainly built on 4 memory spaces.

- *Register file* is a set of physical registers (for instance 16Kb per multiprocessor) split evenly between all active threads of a block.
- *Local memory* is a private space used for per thread temporary data and register spills.
- Each multiprocessor has low-latency on-chip *shared memory* which can be accessed by all threads of a block.

Type	Accessibility	Lifetime
registers	per thread data	thread
local memory	per thread off-chip memory	thread
shared memory	per thread block on-chip memory	block
global memory	all threads as well as CPU	application
CPU memory	not directly accessible by CUDA threads	-

Table B.1: CUDA memory hierarchy

- *Global memory* is visible to all thread blocks of a grid and its lifetime is that of the application.

Table B.1 summarizes these types of memory spaces.

Global memory has no on-chip cache [1] and is of much higher latency than shared memory. Global memory bandwidth is used most efficiently by the following strategy. Observe first that inside a multiprocessor, threads are not scheduled individually; the smallest unit of threads to be scheduled together is called a *half-warp* which consists of 16 threads with consecutive IDs. High throughput is achieved when the simultaneous memory accesses in a half-warp can be coalesced into a single memory transaction of 64, 128 or 256 bytes. The conditions to achieve coalesced accesses are:

1. threads must access 4-byte words, 8-byte words or 16-byte words,
2. all 16 words must lie in the same segment of size equal to the memory transaction size, and
3. threads must access the words in sequence: the k-th thread in the half-warp must access the k-th word. [2]

If a half-warp does not fulfill all the requirements above, a separate memory transaction is issued for each thread and throughput is significantly reduced.

A good usage of shared memory space requires to achieve memory coalescing to the global memory. As illustrated in Figure B.4, threads running in a multiprocessor may reference shared memory in a scattered manner, once data has been loaded from the global memory.

[1] As of 2010, the third generation cards (Fermi series) do provide configurable on-chip cache. Part of shared memory space is served as cache.

[2] The second generation Nvidia cards (GT200 series) are much less restrictive in what concerns the memory access patterns for which coalescing can be achieved. The last requirement has been relaxed as random access within an aligned chunk.

Example 27. *Example 26 only works for a small range n. In this example, we present a general version including data transfer between the global memory and main memory. This illustrates the typical pattern of a heterogeneous CUDA program.*

```
// kernel definition
__global__ void increment_dev(int *X_d, int n)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  X_d[idx] += 1;
}

// Assume n is a multiple of NUM_THREADS = 512
void increment_host(int *X, int n)
{
  // copy X into the global memory of the GPU
  int *X_d;
  cudaMalloc((void**)&X_d, n * sizeof(int));
  cudaMemcpy(X_d, X, n*sizeof(int), cudaMemcpyHostToDevice);

  // setup kernel invocation
  const int NUM_THREADS = 512;
  dim3 nThread(NUM_THREADS, 1, 1);
  dim3 nBlock(n / NUM_THREADS, 1, 1);

  // kernel invocation
  increment_dev<<<nBlock, nThread>>>(X_d, n);

  // copy incremented X_d back to X
  cudaMemcpy(X, X_d, n*sizeof(int), cudaMemcpyDeviceToHost);

  // release the resource
  cudaFree(X_d);
}
```

The above program launches a CUDA kernel with $n/512$ *blocks and each block has* 512 *threads. Each thread resets a single entry of* X*.*

GPU	G80	GT200	Fermi
Cores	128	240	512
SPFP	128 MADD	240 MADD	512 FMA
DPFP	none	30 FMA	256 FMA
Shared Memory	16KB	16KB	48KB or 16KB
L1 Cache	none	none	48KB or 16KB
L2 Cache	none	none	768KB
Concurrent kernels	no	no	up to 16
C++ in device	no	no	yes

Table B.2: Key features of CUDA enabled Nvidia graphics cards

B.4 Performance consideration

CUDA technology is rapidly moving forward in both hardware and software perspective. In Table B.2, we summarize several key features for general purpose CUDA programming. For Nvidia G80 series, there is no double precision floating point computations in hardware. This is due to the fact that single precision is already sufficient for most graphics applications. However, for many non-graphics ones, higher precision floats are much more demanded. The second generation GT200 series start to support double precision floats, but with limited performance, since each multiprocessor in the GPU has only one double precision floating point unit. The third generation Fermi series moves one step further, which fully supports double precision computations in hardware.

Writing runnable parallel programs is usually not very difficult. However, writing efficient parallel code has never been an easy task. This rule holds for CUDA applications too. While attempting to optimizing CUDA code, it is unavoidable to know how to measure the performance accurately and to understand key factors for performance.

$\log_2 n$	memset (MB/s)	cudaMemcpy (MB/s)	cudaMemcpy (MB/s)	memsetKer (GB/s)
23	1600.0	1363.9	1553.3	61.6
24	1600.0	1376.7	1560.5	69.9
25	1422.2	1382.2	1569.0	75.0
26	1422.2	1312.1	1541.5	77.4
27	1462.9	1386.7	1527.2	79.0

Table B.3: Bandwidth tests for clearing a large array

Bandwidth is one of the most important factors for performance, which could be dramatically affected by the choice of memory in which data is stored, and how the

data is accessed. In Table B.3, we list some experimental data collected with functions "memset", "memsetKer", and "cudaMemcpy" to clear a large array of integers. The experiments were completed on a desktop with Intel Core 2 Quad CPU Q9400 @ 2.66GHz and 6GB main memory. The graphics card is the GeForce GTX 285, 1GB global memory with 30×8 SP cores.

The first column of Table B.3 shows the base 2 logarithmic of the array size n. The second column shows the bandwidth in megabytes per second while calling the C function "memset". Third column shows the bandwidth in megabyte per second while calling CUDA API "cudaMemcpy" to move data from main memory to GPU global memory. The fourth column shows the bandwidth in megabyte per second while calling CUDA API "cudaMemcpy" to move data from GPU global memory back to main memory. The last column shows the bandwidth in gigabytes per second of the CUDA kernel shown below, which clears an array residing in GPU global memory with a massive number of threads.

```
__global__ void memsetKer(int *X, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) X[i] = 0;
}
```

Observe that memset has a comparable bandwidth with cudaMemcpy, but their bandwidth is much smaller than that of memsetKer. The theoretical peak bandwidth of this card is 141.6 GB/sec, which is approximately the double of that of memsetKer. From the collected data, one can deduce that CUDA programs should be really data intensive and it is unwise to move small computations into GPUs.

In the CUDA kernel memsetKer, the ith thread accesses the i-th entry of the array. This is the most favorable access pattern to the global memory, where the small accesses are *coalesced* into big accesses. On the contrary, if code accesses the global memory in a scattered or misaligned manner, the performance may downgrade in a significant manner.

Appendix C

Sample CUDA Code

We provide some sample source code for implementing the Stocham FFT.

```
/**
 * @X, input data array of length n = 2^k residing in the device
 * @W, Powers of the primitive root of unity have been precomputed
 * @p, fourier prime number
 *
 * X will be filled with DFT_n(X)
 */
void stockham(int *X, int n, int k, const int *W, int p)
{
  int *Y;
  cudaMalloc((void **)&Y, sizeof(int) * n);

  butterfly(Y, X, k, p);
  for (int i = k - 2; i >= 0; --i) {
      stride_transpose2(X, Y_d, k, i);
      stride_twiddle2(X, W, k, i, p);
      butterfly(Y, X, k, p);
  }
  cudaMemcpy(X, Y, sizeof(int)*n, cudaMemcpyDeviceToDevice);
  cudaFree(Y);
}
```

The function stockham is the top level one, which calls C subroutines:

- butterfly implements $\mathrm{DFT}_2 \otimes I_{2^{k-1}}$,

- stride_transpose2 implements $D_{2,2^{k-i-1}} \otimes I_{2^i}$,
- stride_twiddle2 implements $L_2^{2^{k-i}} \otimes I_{2^i}$.

The following CUDA kernel butterfly_ker implements the subroutine butterfly.

```
/**
 * @X, device array of length n = 2^k
 * @Y, device array of length n = 2^k (output)
 *
 * Y = DFT2 @ I_{2^{k - 1}}(X)
 */
__global__ void butterfly_ker(int *Y, const int *X, int k, int p)
{
  int bid = blockIdx.y * gridDim.x + blockIdx.x;
  int halfn = ((int)1 << (k - 1));
  const int *A = X + bid * blockDim.x;
  int *B = Y + bid * blockDim.x;
  int m = threadIdx.x + halfn;

  B[threadIdx.x] = add_mod(A[threadIdx.x], A[m], p);
  B[m] = sub_mod(A[threadIdx.x], A[m], p);
}
```

This kernel requires that threads in each thread block are indexed only by the x component.

Bibliography

[1] *Magma computational algebra system*. http://magma.maths.usyd.edu.au.

[2] *Nvidia CUDA*. http://www.nvidia.com/object/cuda_home_new.html.

[3] *Nvidia CUDA programming guide 2.3*. 2009.

[4] J. Arnold and R. Gilmer. On the contents of polynomials. *Proc. Amer. Math. Soc.*, 224:556–562, 1970.

[5] M. F. Atiyah and L. G. Macdonald. *Introduction to commutative algebra*. Addison-Wesley, 1969.

[6] P. Aubry, D. Lazard, and M. Moreno Maza. On the theories of triangular sets. *J. Symb. Comp.*, 28(1-2):105–124, 1999.

[7] P. Aubry and M. Moreno Maza. Triangular sets for solving polynomial systems: A comparative implementation of four methods. *J. Symb. Comp.*, 28(1-2):125–154, 1999.

[8] F. Boulier, F. Lemaire, and M. Moreno Maza. Well known theorems on triangular systems. Technical Report LIFL 2001–09, Université Lille I, LIFL, 2001.

[9] F. Boulier, F. Lemaire, and M. Moreno Maza. Well known theorems on triangular systems and the D5 principle. In *Proc. of* Transgressive Computing 2006, Granada, Spain, 2006.

[10] W.S. Brown. The subresultant PRS algorithm. *Transaction on Mathematical Software*, 4:237–249, 1978.

[11] W.S. Brown and J.F. Traub. On Euclid's algorithm and the theory of subresultants. *Jourrnal of the ACM*, pages 505–514, 1971.

[12] D.G. Cantor and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28:693–701, 1991.

[13] S. Chellappa, F. Franchetti, and M. Püschel. How to write fast numerical code. In *Summer School on Generative and Transformational Techniques in Software Engineering*, volume 5235 of *LNCS*, pages 196–259, 2008.

[14] C. Chen, F. Lemaire, O. Golubitsky, M. Moreno Maza, and W. Pan. *Comprehensive Triangular Decomposition*, volume 4770 of *Lecture Notes in Computer Science*, pages 73–101. Springer Verlag, 2007.

[15] C. Chen, F. Lemaire, M. Moreno Maza, W. Pan, and Y. Xie. Efficient computations of irredundant triangular decompositions with the `regularchains` library. In *Proc. of the International Conference on Computational Science (2)*, volume 4488 of *Lecture Notes in Computer Science*, pages 268–271. Springer, 2007.

[16] S.C. Chou and X.S. Gao. On the dimension of an arbitrary ascending chain. *Chinese Bull. of Sci.*, 38:799–804, 1991.

[17] G.E. Collins. Subresultants and reduced polynomial remainder sequences. *Journal of the ACM*, 14:128–142, 1967.

[18] G.E. Collins. The calculation of multivariate polynomial resultants. *Journal of ACM*, 18(4):515–532, 1971.

[19] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp.*, 19:297–301, 1965.

[20] A. Corso, W. V. Vasconcelos, and R. H. Villarreal. On the contents of polynomials. *J. Pure. Appl. Algebra*, 125(1-3):117–127, 1998.

[21] M. Şahin. On the minimal number of elements generating an algebraic set. Master thesis, Bilkent University, 2002.

[22] X. Dahan, M. Moreno Maza, É. Schost, and Y. Xie. On the complexity of the D5 principle. In *Proc. of* Transgressive Computing 2006, Granada, Spain, 2006.

[23] X. Dahan and É. Schost. Sharp estimates for triangular sets. In *ISSAC 04*, pages 103–110. ACM, 2004.

[24] J. Della Dora, C. Dicrescenzo, and D. Duval. About a new method for computing in algebraic number fields. In *Proc. EUROCAL 85 Vol. 2*, volume 204 of *Lect. Notes in Comp. Sci.*, pages 289–290. Springer-Verlag, 1985.

[25] L. Ducos. Optimizations of the subresultant algorithm. *Journal of Pure and Applied Algebra*, 145:149–163, 2000.

[26] P. Duhamel and M. Vetterli. Fast Fourier transforms - A tutorial review and a state-of the art. *Signal Processing*, 4(19):259–299, 1990.

[27] J.-G. Dumas, T. Gautier, and C. Pernet. Finite field linear algebra subroutines. In *ISSAC '02: Proceedings of the 2002 international symposium on symbolic and algebraic computation*, pages 63–74, New York, NY, USA, 2002. ACM.

[28] D. Duval. *Questions relatives au calcul formel avec des nombres algébriques*. Université de Grenoble, 1987. Thèse d'État.

[29] D. Eisenbud. *Commutative algebra*. Springer, Springer-Verlag, 1994.

[30] M. El Kahoui. An elementary approach to subresultants theory. *J. Symb. Comp.*, 35:281–292, 2003.

[31] A. Filatei, X. Li, M. Moreno Maza, and É Schost. Implementation techniques for fast polynomial arithmetic in a high-level programming environment. In *Proc. ISSAC'06*, pages 93–100, New York, NY, USA, 2006. ACM Press.

[32] F. Franchetti and M. Püschel. *Encyclopedia of parallel computing*, chapter Fast Fourier Transform. Springer, 2011.

[33] F. Franchetti, M. Püschel, Y. Voronenko, S. Chellappa, and J. M. F. Moura. Discrete Fourier transform on multicore. *IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores"*, 26(6):90–102, 2009.

[34] J. von zur Gathen and J. Gerhard. *Modern computer algebra*. Cambridge University Press, 1999.

[35] T. Gómez Díaz. *Quelques applications de l'évaluation dynamique*. PhD thesis, Université de Limoges, 1994.

[36] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete Fourier transforms on graphics processors. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[37] GPUmat. *GPU toolbox for MATLAB*.

[38] J. van der Hoeven. Truncated Fourier transform. In *Proc. ISSAC'04*. ACM Press, 2004.

[39] É. Hubert. Notes on triangular sets and triangulation-decomposition algorithms. I. Polynomial systems. In *Symbolic and numerical scientific computation (Hagenberg, 2001)*, volume 2630 of *LNCS*, pages 1–39. Springer, 2003.

[40] F. Ischebeck and R. A. Rao. *Ideals and reality, projective modules and number of generators of ideals*. Springer-Verlag, 2005.

[41] M. Kalkbrener. A generalized Euclidean algorithm for computing triangular representations of algebraic varieties. *J. Symb. Comp.*, 15:143–167, 1993.

[42] M. Kalkbrener. *Algorithmic properties of polynomial rings*. Dep. of math., Swiss Federal Institute of Technology, Zurich, 1995. Habilitation Thesis.

[43] M. Kalkbrener. Algorithmic properties of polynomial rings. *J. Symb. Comp.*, 26(5):525–581, 1998.

[44] D. Lazard. A new method for solving algebraic systems of positive dimension. *Discr. App. Math*, 33:147–160, 1991.

[45] D. Lazard. Solving zero-dimensional algebraic systems. *J. Symb. Comp.*, 15:117–132, 1992.

[46] F. Lemaire, M. Moreno Maza, W. Pan, and Y. Xie. When does $\langle T \rangle$ equal $\mathsf{sat}(t)$? *J. of Symbolic Computation*, to appear.

[47] F. Lemaire, M. Moreno Maza, W. Pan, and Y. Xie. When does $\langle T \rangle$ equal $\mathsf{sat}(t)$? In *Proc. ISSAC'20008*, pages 207–214. ACM Press, 2008.

[48] F. Lemaire, M. Moreno Maza, and Y. Xie. The `RegularChains` library. In Ilias S. Kotsireas, editor, Maple Conference 2005, pages 355–368, 2005.

[49] X. Li, M. Moreno Maza, and W. Pan. Computations modulo regular chains. In *In ISSAC'09*, pages 239–246. ACM Press, 2009.

[50] X. Li, M. Moreno Maza, R. Rasheed, and É. Schost. The modpn library: Bringing fast polynomial arithmetic into maple. In *MICA'08*, pages 73–80, 2008.

[51] X. Li, M. Moreno Maza, R. Rasheed, and É. Schost. The modpn library: Bringing fast polynomial arithmetic into maple, 2008. Submitted to J. of Symbolic Computation.

[52] X. Li, M. Moreno Maza, and É Schost. Fast arithmetic for triangular sets: From theory to practice. In *Proc. ISSAC'07*, pages 269–276, New York, NY, USA, 2007. ACM Press.

[53] X. Li, M. Moreno Maza, and É. Schost. Fast arithmetic for triangular sets: From theory to practice. *J. Symb. Comp.*, 44(7):891–907, 2008.

[54] Project LinBox. *Exact computational linear algebra*. http://www.linalg.org/.

[55] C. Van Loan. *Computational frameworks for the fast Fourier transform*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[56] MathWorks. *Parallel computing toolbox*.

[57] M. Moreno Maza and W. Pan. Fast polynomial multiplication on a GPU. In *HPCS'10, Proceedings of the High Performance Computing Symposium*, 2010.

[58] B. Mishra. *Algorithmic algebra*. Springer-Verlag, New Yor, 1993.

[59] M Monagan. Probabilistic algorithms for computing resultants. In *ISSAC '05: Proceedings of the 2005 international symposium on Symbolic and algebraic computation*, pages 245–252, New York, NY, USA, 2005. ACM.

[60] M. Monagan and R. Pearce. Rational simplification modulo a polynomial ideal. In *ISSAC '06*, pages 239–245. ACM Press, 2006.

[61] M. Monagan and R. Pearce. Parallel sparse polynomial multiplication using heaps. In *ISSAC '09: Proceedings of the international symposium on Symbolic and algebraic computation*, pages 263–270, New York, NY, USA, 2009. ACM.

[62] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[63] K. Moreland and E. Angel. The FFT on a GPU. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, Aire-la-Ville, Switzerland, 2003. Eurographics Association.

[64] M. Moreno Maza. Integration of triangular sets methods in Aldor. Technical report, NAG Ltd, Oxford, UK, 1998. FRISCO task 3.3.2.4.2.

[65] M. Moreno Maza. On triangular decompositions of algebraic varieties. Technical Report TR 4/99, NAG Ltd, Oxford, UK, 1999. Presented at the MEGA-2000 Conference, Bath, England.

[66] M. Moreno Maza and W. Pan. Solving bivariate polynomial systems on a GPU. Technical report, Universit of Western Ontario, 2011.

[67] M. Moreno Maza and R. Rioboo. Polynomial gcd computations over towers of algebraic extensions. In *Proc. AAECC-11*, pages 365–382. Springer, 1995.

[68] M. Moreno Maza and Y. Xie. Balanced dense polynomial multiplication on multicores. In *Proc. of PDCAT'09*. IEEE Computer Society, 2009.

[69] M. Moreno Maza and Y. Xie. FFT-based dense polynomial arithmetic on multicores. In D.J.K. Mewhort, editor, *Proc. HPCS 2009*, volume 5976 of *LNCS*, Heidelberg, 2010. Springer-Verlag Berlin.

[70] P. N.Swarztrauber. FFT algorithms for vector computers. *Parallel Comput.*, 1(1):45–63, 1984.

[71] V. Y. Pan. Simple multivariate polynomial multiplication. *J. Symb. Comp.*, 18(3):183–186, 1994.

[72] M. Püschel, F. Franchetti, and Y. Voronenko. *Encyclopedia of parallel computing*, chapter Spiral. Springer, 2011.

[73] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[74] J. F. Ritt. *Differential algebra*. Amer. Math. Soc, New York, 1950.

[75] G. Ruetsch and P. Micikevicius. Optimizing matrix transpose in CUDA. 2009.

[76] É. Schost. Complexity results for triangular sets. *J. Symb. Comp.*, 36(3-4):555–594, 2003.

[77] V. Shoup. *NTL: the number theory library.*

[78] T. G. Stockham. High-speed convolution and correlation. In *AFIPS '66 (Spring): Proceedings of the April 26-28, 1966, Spring joint computer conference*, pages 229–233, New York, NY, USA, 1966. ACM.

[79] B. Sturmfels. *Solving systems of polynomial equations*. Amer. Math. Soc., 2002.

[80] T. Coquand, L. Ducos, H. Lombardi, and C. Quitté. On the contents of polynomials. *Revue des Mathématiques de l'enseignement Supérieur*, 113(3):25–39, 2003.

[81] J. von zur Gathen and J. Gerhard. Fast algorithms for Taylor shifts and certain difference equations. In *ISSAC'97*, pages 40–47. ACM Press, 1997.

[82] D. M. Wang. Computing triangular systems and regular systems. *J. Symb. Comp.*, 30(2):295–314, 2000.

[83] W. T. Wu. On zeros of algebraic equations – an application of Ritt principle. *Kexue Tongbao*, 31(1):1–5, 1986.

[84] L. Yang and J. Zhang. Searching dependency between algebraic equations: an algorithm applied to automated reasoning. Technical Report IC/89/263, International Atomic Energy Agency, Miramare, Trieste, Italy, 1991.

[85] C.K. Yap. *Fundamental problems in algorithmic algebra*. Princeton University Press, 1993.

Curriculum Vitae

Name: **Wei Pan**

Post-Secondary Education and Degrees:

The University of Western Ontario
London, Ontario, Canada
Ph.D. Computer Science, December 2010

University of Science & Technology of China
Hefei, Anhui, China
M.Sc. in Mathematics, July 2006

University of Science & Technology of China
Hefei, Anhui, China
B.Sc. in Mathematics, July 2003

Work Experience:

Internship
Maplesoft incorporation, Waterloo, Ontario, Canada.
May 2010 - Aug. 2010

Research Assistant & Teaching Assistant.
University of Western Ontario, London, Canada.
Sept. 2006 - Dec. 2010

Research Assistant & Teaching Assistant.
University of Science & Technology of China,
Hefei, Anhui, China.
Sept. 2003 - Jul. 2006

Publications:

Fast polynomial multiplication on a GPU
With Marc Moreno Maza
Proceedings of the High Performance Computing Symposium (HPCS2010).

When does $\langle T \rangle$ equal $\mathsf{sat}(T)$?
With Francois Lemaire, Marc Moreno Maza and Yuzhen Xie
Journal of Symbolic Computation, to appear.

Computation modulo regular chains
With Xin Li and Marc Moreno Maza
Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC 2009).

When does $\langle T \rangle$ equal $\mathsf{sat}(T)$?
With Francois Lemaire, Marc Moreno Maza and Yuzhen Xie
ISSAC 2008.

On the verification of polynomial system solvers
With Changbo Chen, Marc Moreno Maza and Yuzhen Xie
Frontiers of Computer Science in China, Vol 2, n.o. 1, 2008.

The `ConstructibleSetTools` and `ParametricSystemTools` modules of the `RegularChains` library in Maple.
With Changbo Chen, Francios Lemaire, Liyun Li, Marc Moreno Maza and Yuzhen Xie
Proceedings of the International Conference on Computational Science and Applications, IEEE Computer Society, pages 342-352, 2008.

Publications:

Comprehensive triangular decomposition
With Changbo Chen, Francios Lemaire, Oleg Golubitsky and Marc Moreno Maza
Computer Algebra in Scientific Computing (CASC 2007).

Efficient Computations of Irredundant Triangular Decompositions with the `RegularChains` Library
with Changbo Chen, Francois Lemaire, Moreno Moreno Maza and Yuzhen Xie
Proceedings of Computer Algebra Systems and Their Applications, LNCS 4488, pp. 268-271, 2007.

Uniform Groebner bases for ideals generated by polynomials with parametric exponents
With Dongming Wang
ISSAC 2006.